AD-A043 452 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
ON THE DESIGN AND VERIFICATION OF OPERATING SYSTEMS.(U)
MAY 77 L FLON F44620-73-C-0074

UNCLASSIFIED AFOSR-TR-77-1139 NL

1 OF 2
AD
A043452

AFOSR-TR- 77 - 1139

7

β.S.

On the Design and Verification of Operating Systems

Lawrence Flon

May 1977

Approved for public release;
distribution unlimited.

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is
approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 AFOSR-TR- 77-1139	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 ON THE DESIGN AND VERIFICATION OF OPERATING SYSTEMS.	9	5. TYPE OF REPORT & PERIOD COVERED Interim rept. 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Lawrence Flon		8. CONTRACT OR GRANT NUMBER(s) 15 F44620-73-C-0074 ✓ NSF-DCR-74-24573
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213	16	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2 17 A2
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE 11 May 1977 13. NUMBER OF PAGES 100
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research Bolling AFB, DC 20332	12 107 p.	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis applies and extends mathematical program verification to systems programs. The thesis is both methodological (in proposing a methodology for the design and verification of large programs), and theoretical (in presenting various results dealing with the correctness of parallel programs).

next page

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403 081

20. ABSTRACT (Continued)

NO 4

The design methodology is based upon the use of *abstract data types* and the construction and verification of both specifications and implementations for them. The abstract data type is a means of modularization which encapsulates the representation of a data structure and the algorithms which operate directly upon it. The specification technique appeals to various mathematical structures (e.g. sets and sequences) to describe an abstract state for objects of a given type. The correctness of the formal specifications is cast in terms of the proof of certain invariant properties of the abstract state. An axiomatic proof rule is given to formulate the theorems necessary for proving the invariance of predicates across formal specifications.

The applicability of the methodology to operating systems is explored. It is found that a hierarchical decomposition is most amenable to verification, and that the implementation language used is a function of that hierarchy. The example of a process dispatcher module of a hypothetical operating system is used to illustrate the process of design, specification, implementation, and verification using the methodology. Various properties are proven of the abstract specifications, including one representation of the concept of fair service. Programs are then written for the specifications and their correctness is verified.

Three different approaches to the total correctness of parallel programs are treated. The first uses the weakest pre-condition concept to explore statically the combinatoric interactions which may occur among parallel programs during execution. The method is complete but computationally complex. A second approach extends the axiomatic weak correctness results of Owicki to include a technique for proof of loop termination. The concept of a *steady state loop invariant* is introduced and used to establish the total correctness of an old scheme of mutual exclusion which appeals only to the indivisibility of memory access for synchronization.

The third approach treats a syntactically restricted class of parallel programs. For this class we give definitions for the weakest pre-conditions which guarantee weak correctness and absence of blocking, deadlock, and starvation. We also formulate theorems which use invariant assertions to circumvent the actual weakest pre-condition computation.

On the Design and Verification of Operating Systems

Lawrence Flon

May 1977

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported in part by the Defense Advanced Research Projects Agency under contract no. F44620-73-C-0074, and in part by the National Science Foundation under grant DCR74-24573, and monitored by the Air Force Office of Scientific Research.

Abstract:

This thesis applies and extends mathematical program verification to systems programs. The thesis is both methodological (in proposing a methodology for the design and verification of large programs), and theoretical (in presenting various results dealing with the correctness of parallel programs).

The design methodology is based upon the use of *abstract data types* and the construction and verification of both specifications and implementations for them. The abstract data type is a means of modularization which encapsulates the representation of a data structure and the algorithms which operate directly upon it. The specification technique appeals to various mathematical structures (e.g. sets and sequences) to describe an abstract state for objects of a given type. The correctness of the formal specifications is cast in terms of the proof of certain invariant properties of the abstract state. An axiomatic proof rule is given to formulate the theorems necessary for proving the invariance of predicates across formal specifications.

The applicability of the methodology to operating systems is explored. It is found that a hierarchical decomposition is most amenable to verification, and that the implementation language used is a function of that hierarchy. The example of a process dispatcher module of a hypothetical operating system is used to illustrate the process of design, specification, implementation, and verification using the methodology. Various properties are proven of the abstract specifications, including one representation of the concept of fair service. Programs are then written for the specifications and their correctness is verified.

Three different approaches to the total correctness of parallel programs are treated. The first uses the weakest pre-condition concept to explore statically the combinatoric interactions which may occur among parallel programs during execution. The method is complete but computationally complex. A second approach extends the axiomatic weak correctness results of Owicki to include a technique for proof of loop termination. The concept of a *steady state loop invariant* is introduced and used to establish the total correctness of an old scheme of mutual exclusion which appeals only to the indivisibility of memory access for synchronization.

The third approach treats a syntactically restricted class of parallel programs. For this class we give definitions for the weakest pre-conditions which guarantee weak correctness and absence of blocking, deadlock, and starvation. We also formulate theorems which use invariant assertions to circumvent the actual weakest pre-condition computation.

Acknowledgements

My primary source of inspiration and direction over the years has of course been my advisor, Nico Habermann. He has provided me with a role model for advising which I shall strive to live up to for the sake of my own future students. My thanks also to Nori Suzuki, who is responsible for the theoretical leaning this thesis took when he joined my committee. I would like to express my gratitude to Bill Wulf, both for his comments on the various drafts and for acting as pseudo-advisor during the semester that Nico was away on sabbatical. Thanks also to Peter Andrews, for his role as "outside" committee member.

I am pleased to acknowledge the incredibly careful job that Sten Andler did in reading the first draft.

1. Introduction

The primary motivation for the work contained in this thesis is the desire to apply and extend mathematical program verification to the realm of systems programs. It has long been recognized that the testing and debugging of large programs cannot instill much confidence in their correctness, since it is rarely possible to cause large programs to execute all possible paths. Thus there has been a steadily increasing amount of research done to discover methods by which programs may be statically verified - i.e. deemed correct without any execution at all.

It is not our contention that run-time debugging should be totally abandoned in favor of correctness proofs, since any verification of a program, be it mathematical or otherwise, requires the user of that program to take at least something on faith. In the case of debugging, we must believe that all possible inputs (or at least a covering set of them) have been checked against their outputs, and that the results conform to those predicted by the program specifications. In the case of mathematical verification, we must believe that all necessary theorems have been proven, and we must believe those proofs. In either case we must believe that the program specifications are themselves correct.

Of late, our inherent lack of confidence in both of these verification methods has led to their (at least semi-) automation, so that programs now exist to debug other programs by the generation of appropriate test data, and there are program verifiers which attempt to generate and prove the theorems necessary for establishing the correctness of other programs. Of course, we have merely pushed the problem off a little, since we are now faced with believing that the automatic testers and verifiers are correct, but at the same time we somehow gain confidence in the ultimate correctness of the target programs.

The result is that although we cannot in general be absolutely certain that our large programs are correct, the more well-defined verification techniques we apply, the more confidence we gain, and confidence is absolutely necessary in systems which perform critical services.

Returning to the stated purpose of our research with some conviction as to its usefulness, we find, alas, that really large programs are way beyond the reach of known techniques of mathematical verification. This is due to the vast increase with program size of both the number and complexity of required theorems. Although the automation of mathematical verification mentioned previously is both necessary and useful, conventional verifiers [Suzuki 75, Good 75] often cannot handle the complexity. Thus we find that large programs, which are not verifiable by testing, are likewise not verifiable mathematically, and small programs, which are for the most part completely testable, are the only ones which can be completely proved correct. This has not only caused research in mathematical verification to be scoffed at by industry, but has resulted in the discouragement of many researchers.

Part of this thesis then, Chapter 2, deals with a methodology for verifying large programs by making them appear to be small programs and applying known techniques. At this point we might mention that those who are interested in verification (from here on we will drop the prefix "mathematical") fall roughly into two camps - those who are interested in techniques to verify arbitrary programs, and those, like us, who will settle for verifying programs which were constructed with verification in mind. It is our firm belief that all programs can be so constructed, and that their resulting performance, especially in view of the ever-faster hardware being built, can be made to be very close to that of programs written with only performance in mind.

We have thus far invoked the notion of a "correct" program several times, appealing to an intuitive understanding of the term. The meaning of correct we have in mind requires something to compare against - i.e. the "right" answer. It is very common when debugging a program to find that the wrong answers it gives are due to an unsatisfactory or inconsistent definition of correctness. Thus we can have a correct program (with respect to a given definition of correctness) which is completely useless for the problem at hand. What is needed is a description of what the program must do in a form other than that of the program text itself, so that we may operate on that description to verify it without having to appeal to the particular implementation choices made by an actual program. In practice the description often takes the shape of prose, but prose does not lend itself to mathematical analysis. In Chapter 2, we discuss the related problems of constructing formal specifications and transforming programs into something that can be conveniently compared with them. Furthermore, in Chapter 2 we discuss an approach to defining the correctness of formal specifications themselves, and provide a means to verify that correctness.

In Chapter 3, we are concerned with the application of the methodology of Chapter 2 to operating systems. We first examine the relative effects of system structure, implementation language, and verification on one another. Subsequently we present the design, implementation, and verification of a process dispatcher, paying attention not only to proof of implementation, but also to proof of specifications as outlined in Chapter 2.

Operating systems employ a good deal of concurrency, and satisfactory techniques for verifying parallel programs are not yet known, although work is beginning to be done on the subject. Owicki [Owicki 75, Owicki 76] has extended the work of Hoare [Hoare 71a] so that the insightful addition of auxiliary variables to parallel programs will allow them to be verified. Griffiths [Griffiths 74] constructed a system which could verify certain properties of parallel programs, although the method depends heavily upon the properties of the ECL [Prenner 72] process controller. Habermann [Habermann 72], Flon and Habermann [Flon 76], Howard [Howard 76], and Saxena [Saxena 76] have all attacked the problem of verification applied to particular synchronization mechanisms from a data- rather than process-oriented viewpoint. In Chapter 4 we explore two different approaches to verifying the total correctness of parallel programs. The first uses Dijkstra's weakest pre-condition semantics [Dijkstra 76] to consider the combinatoric interactions which may occur during program execution, and results in a complete but computationally difficult method of verification. The second approach is an extension of Owicki's methodology applied to arbitrary programs, which facilitates proofs of strong correctness (loop termination).

Neither of the two approaches of Chapter 4 appear very promising for automation. Since the verification of a large-scale system is guaranteed to be a large project, any help which can be provided via automation is welcome. While we treat a fairly large class of parallel programs in Chapter 4 without much automatable success, in Chapter 5 we discuss the problem of parallel program verification for a restricted syntactic category of programs, and present well-defined, automatable methods for verifying the weak correctness of terminating parallel systems (i.e. verifying their output), and the strong correctness of non-terminating, cyclic parallel systems. In particular, we give a formal definition of the weakest-pre-condition of a cyclic parallel program that guarantees absence of blocking, deadlock, and starvation. We also present a verification method for strong correctness, along the lines of the invariant method of sequential loop verification.

Chapter 6 contains a summary of the results, along with an identification of the contributions this thesis makes to the field. We also evaluate the feasibility of completely verifying a useful operating system, and identify what areas require more research to make it practical.

2. Methodology

2.1. Programming Language

The programming language in which a program is written plays a vital part in determining the amount of effort required to verify its correctness. We shall see later that the specification language used to express the verification goals plays an equally vital part. For the moment we will restrict ourselves to the former.

2.1.1 Level

In discussing the most appropriate language features for the implementation and verification of large-scale systems, there are many considerations to be taken into account. Among the most important are expressiveness, verifiability, and efficiency. Historically, the first two have been thought by many to be in conflict with the third, but this is not the case. For a long time, people were unwilling to write systems programs in anything other than assembly language. This was due both to the inability of compilers for high-level languages to generate efficient machine code, and to the inappropriateness of language features for the problem at hand. However, since the early years of compiler development, work on optimization has enabled compilers to generate code which is often better than that produced by experienced systems programmers (e.g. [Wulf 75]). There is no question as to the advantages of high-level languages for verification. This is due in large part to the unprotected nature of most assembler instructions with respect to the type and scope of values which they may change. The Algol 60 concept of restricted scope, for example, is an invaluable asset when it comes to verifying assertions of the form "and there is no other way to change this value."

Better expressiveness is not quite so easily attributed to high-level languages, but it will be seen in succeeding sections that the abstraction features which a compiler can provide play a crucial role.

2.1.2 Modularity

We can subsume the concept of scope in that of modularity, since if the scope of a variable or group of variables is to be limited to only part of a program, then that part is separable from the rest. Exactly what constitutes a module is the subject of this section.

The first thing we must ask is, "What are the desirable attributes of a good modular decomposition?" One is that each module should constitute a separate work assignment, so that small groups of programmers can work independently of one another (the "too many cooks" syndrome). Another is that most reasonable changes made to the system should require explicit alteration of as few modules as possible. While one way to achieve the latter would be to limit ourselves to one module, this is effectively prohibited by the former. A more reasonable way to achieve localization of changes is to assign to each module the responsibility for implementing some design decision. The criteria for deciding what constitutes separable design decisions are certainly open to discussion (see [Parnas 72b]), but an extremely useful principle is to associate each module with the implementation and management of a class or data structures. Modularizing along the lines of data structure classes results in small, contained sub-programs, and therein lies the benefit to verification. When assertions about the properties of a data structure can be verified once and for all by only considering the implementation of a few accessing operations, then those "invariant relations" can be assumed as given when verifying programs which use those operations. Otherwise it is necessary to take into account every possible program statement which may change the structure, and there may be very many. The complexity of theorems to be proven decreases to a more manageable size if code to actually alter a structure occurs in only a few places.

This principle plays an important role in the methodology we propose for verifying large programs, and manifests itself in various ways. We state it more precisely here, chiefly for emphasis, naming it the Principle of Maximal Encapsulation:

> The usage of a program module should never affect its correctness (although it affects that of the user). The correctness of a system decomposed into modules is then determined solely by the collective correctness of the individual modules.

If the principle of maximal encapsulation is adhered to in the decomposition of a system, then the verification of that system should consist of a separate, independent, verification for each module. It is only by this application of the divide and conquer strategy that we can hope to verify a large-scale system. The onus is then placed on the designers to create a good decomposition, but that is, after all, where it belongs.

2.1.3 Abstraction

Having decided to modularize a system on the basis of the implementation of data structure classes, we must now decide on the representation of a module in a programming language designed to support verifiable systems programs. Since the purpose of a module is to hide the details of the implementation of a data structure, that module represents an abstraction to its users. This abstraction takes a behavioral form, as in for example describing the properties of a stack without considering whether it is implemented as a list or an array. One of the earliest examples of linguistic abstraction in programming is the subroutine. The subroutine as it existed even in the earliest version of FORTRAN provides the programmer with the means to 1) invoke the same program segment from several places, thus saving multiple coding; 2) parametrize that program segment so that several similar but slightly different segments could be collapsed into one; 3) defer the coding of a program segment until necessary, or to permit someone else to work on it; 4) hide the details of an algorithm from a programmer so as not to have them interfere in his own task.

The subroutine (or procedure as it came to be known in Algol 60) is clearly an important concept, but is not in itself sufficient to provide the modularity we desire. Because we want modules to restrict data structure access, we must not allow those data structures to be declared in such a way as to permit direct access by other than the "privileged" code of a certain procedure. In basic Algol 60 syntax, the data must therefore be declared local to the procedure which will operate upon it. This cannot be done when more than one procedure must have the direct-access privilege. To solve the problem, we need to be able to "encapsulate" the data declarations along with the applicable procedures in one textual unit.

The first attempt at this kind of encapsulation was the class concept of Simula 67 [Dahl 68]. There has been much work since which has refined that approach. A discussion of the basic issues may be found in [Liskov 74], and in [Flon 74] along with a comparison of Simula 67, Algol 68, and Pascal along these lines. Current research efforts include [Wulf 76, Schaffert 75, Popek 77, Ambler 77, Geschke 77, Johnson 76].

The refined syntactic mechanism has come to be known as the *abstract data type*. *Type* is an equivalence relation which partitions program data into classes based upon the operations applicable to each class. The abstract data type provides the means to accomplish a very natural extension of the set of data types pre-defined in a language. These pre-defined types usually include at least *integer* and *boolean*, and possibly *real*. Each of the so-called "primitive" types is characterized by a set of abstract values, such as the *boolean* values *TRUE* and *FALSE*, in addition to type-specific operations, such as the *boolean* functions *AND*, *OR*, and *NOT*.

To extend the set of types, a programmer must describe both the behavior and internal representation of objects of the new type. The implementation details, namely the representation used (which we will call the structure` and the algorithms which implement the operations, are the design decisions which the new data type hides from users. A description of the abstract values and the behavior of the operations is provided to users through the module specifications.

As a brief example, consider the extension of the set of types with the new type "complex", which is intended to model the behavior of values in the complex plane. The abstract values, then, are ordered pairs, and the behavior of such operations as complex addition or multiplication is the same as vector arithmetic. A type definition in our language will take the form:

```
type typename =
        <structure>
        <operations>
endtype
```

If we had such a definition for "complex", we would be able to use it in a subsequent program as follows:

```
var x,y,z: complex;

x←complex.create(1,2);
y←complex.create(2,-3);
z←complex.mul(x,y);
```

Here we assume that the call "complex.create(a,b)" creates an object of type complex with abstract value a+bi by invoking the "create" operation of the type, and that the call "complex.mul(x,y)" invokes the "mul" operation of type complex on the values x and y, returning the product. Then x=1+2i, y=2-3i, and z=8+i afterwards (if the implementation of "complex" is correct). One definition of "complex" might be

```
type complex =
        var re:real, im:real;

        op mul(x:complex,y:complex):complex =
                complex.create(x.re*y.re-x.im*y.im, x.re*y.im+x.im*y.re);
        .
        .
        .
endtype
```

while another might be

```
type complex =
      var rho:real, theta:real;

      op mul(x:complex,y:complex):complex =
            complex.create(x.rho*y.rho, x.theta+y.theta);
      .
      .
      .
endtype
```

In this way we are able to hide from users both the representation (rectangular or polar coordinates) and the algorithms.

Using abstract data types as our *modularization tool*, we have a means to adhere to the principle of maximal encapsulation. Since it is possible to directly alter an object only from within the scope of that object's type definition, only the operations defined therein can effect that alteration. If the operations defined are not too primitive (i.e. if they don't defeat our whole purpose by allowing arbitrary assignment to critical subfields) and if they are careful to check their parameters, it should be possible to have absolute faith in the consistency of the data structure. Subsequent sections will discuss the concept of data structure consistency more formally.

2.2. Specification Language

The question of how to write formal specifications for the abstract data types which comprise a system is very important, for it impacts not only the difficulty of constructing those specifications, but also the complexity of verification - both the verification of a particular data type implementation and that of "global" properties of the system. An example of the latter distinction is the difference between verifying that a ready-process queue always has a valid queue structure, and verifying that processes on the queue are serviced in Round Robin fashion. We will examine three specification techniques here which seem currently to be the most promising. Liskov and Zilles [Liskov 76] present several specification techniques along with a similar type of analysis, although we shall draw some different conclusions from theirs. We disagree also on the classification scheme used.

The three methods can all be classified as "axiomatic", in that in each case, a complete specification consists of a finite set of rules which 1) sufficiently describe the desired properties of an object in a manner which in general allows for many different implementations (in the same way that a given set of axioms of first-order logic allows many different interpretations), and 2) when combined with rules of inference, provide the means for deductive proofs of theorems. The axiomatic methods are widely differing though, and this can be attributed to differences in form as well as rules of inference. We further classify the methods as 1) algebraic, 2) state machine oriented, and 3) predicate transforming.

2.2.1 Algebraic Specifications

Algebraic specifications have been examined by Guttag [Guttag 75] and discussed in [Liskov 76]. The term "algebraic" stems from a formal basis in heterogeneous algebra. The reader who is interested in further details is referred to [Guttag 75]. From a practical point of view, the algebraic specifications of a data type consist of a domain-range description of its operators, and a series of axioms which define those operators in terms of their relationships to one another. As an example consider the algebraic specification of a type "queue" (in fact a FIFO queue) of elements of some other type t, similar to that presented in [Guttag 75].

Domains and Ranges:

newq: → queue	<create an empty queue>
enq: queue X t → queue	<create a queue with an element appended to tail>
deq: queue → queue	<create a queue with an element removed from head>
first: queue → t	<return value of head element>
empty: queue → boolean	<test for empty queue>

Axioms: q:queue, k:t

empty(newq) = true
empty(enq(q,k)) = false
first(newq) = error
first(enq(q,k)) = if empty(q) then k else first(q)
deq(newq) = error
deq(enq(q,k)) = if empty(q) then newq else enq(deq(q),k)

Each axiom equates a call upon one of the operations with a primitive recursive function. The set of axioms comprises a "sufficiently complete" description of the notion of FIFO queue. For our purposes we will take this to mean "all necessary cases are accounted for" (see [Guttag 75] for a more formal definition of sufficient completeness and for a discussion of its proof). The presence of the term "error" in the specifications is a means of expressing those cases which should never occur in any program correctly using these specifications.

The algebraic specification technique exhibits a pleasing structure in this case, as it does in many other such examples. One of its deficiencies is the purely mathematical nature of the functions which are specified. In the mathematical sense, functions do not cause state changes. In particular, there is no way to express that the deq operation should remove a queue element and return that element as its value. This problem is inherent in the formal basis. It has been proposed [Guttag 77] that the mechanism be extended with the definition of composite functions, such as

deqfirst(x,q) = x←first(q); q←deq(q)

although this appears to be somewhat ad hoc.

A second problem with the technique is reflected by the non-existence of axioms which define enq explicitly, and can be seen in the specification of a bounded queue:

```
length: queue → integer

length(newq) = 0
length(enq2(q,k)) = 1+length(q)
deq(newq) = error
deq(enq2(q,k)) = if empty(q) then k else enq2(deq(q),k)
enq(q,k) = if length(q) ≥ m then error else enq2(q,k)
```

Here we are forced to invent an auxiliary operator, enq2, in order to allow the detection of "error" upon insertion of a new element in a full queue. In addition to the lack of intuitive interpretations for these new operators, a programmer may be mislead into believing them to be necessary in the implementation. The reader should convince himself that if such an operation is not used, then "error" can only be detected at the point a "more than full" queue is used for examining or dequeueing.

These problems detract somewhat from the pleasing mathematical elegance of the technique, but they are not really sufficient evidence upon which to base our evaluation. There are two other problems that seem more serious from our point of view. The first deals with the issue of specifying the behavior of objects which are used by concurrent processes. The algebraic technique provides no help for us in this problem simply because the purely functional nature of the specifications admits no notion of the interference and non-determinacy introduced by concurrency. The other problem is associated with the verification of particular implementations for algebraic specifications. The basic idea in such a verification is to examine each axiom in turn, and prove that the left and right sides compute the same function when the programs are substituted for the abstract operators. For example, if an axiom for type "boolean" were

(1) imp(a,b) = or(not(a),b)

and the programs were:

```
imp(a,b) = if a=0 then 1 else b fi;
or(a,b) = if a=1 then 1 else b fi;
not(a) = if a=0 then 1 else 0 fi;
```

then or(not(a),b) becomes

```
     (2)  if (if a=0 then 1 else 0 fi)=1 then 1 else b fi

or   (3)  if a=0 then 1 else b fi
```

which is imp(a,b), and axiom (1) would be verified.

The problem with this strategy is that it relies heavily on program transformation such as that between (2) and (3) above. For most real-life examples, the substituted programs do not transform into one another easily, as the reader would discover in attempting to verify an Algol-like implementation of type "queue". Additionally, each operator occurs in several places in the axioms, and therefore its implementation program must be combined with other programs many times, which gives one the feeling that the same program must be verified over and over again (in actual fact, smaller and different aspects of the programs are really being verified at each stage, but that does not make it proportionally easier).

Finally, users of the algebraic specification technique admit to the fact that constructing such axioms for non-trivial types is difficult, although they claim that experience does help. There is some question in our mind as to why the difficulty should exist. We feel that most of the problem is due to the fact that the way we visualize structures is significantly different from the algebraic approach, and we would therefore prefer a mechanism with which it is easier to translate our thoughts into mathematics.

2.2.2 State Machine Model

The state machine model for specification is due to Parnas [Parnas 72a], and was adopted by the group at SRI concerned with the design of a provably secure operating system [Neumann 74]. It is based on the notion that an object is controlled by both passive (information gathering) and active (state transforming) functions. The passive functions, called V-functions (V for value), have no side effects whatsoever. They simply return information about the current state of an object. The active functions, called O-functions (O for operation), change the state by altering the values of V-functions. The only relevant program state is in fact contained in the values of all the V-functions. In the case of the bounded queue, such specifications would be:

k:t, *j*,*l*:integer

V-function: *k*←*first*

purpose: returns first queue element
initially: undefined
exceptions: length=0

V-function: *l*←*length*

purpose: returns length of queue
initially: 0
exceptions: none

hidden V-function: *k*←*el(j)*

purpose: returns j'th queue element
initially: undefined
exceptions: $j<0 \vee j\geq \text{length}$

O-function: *enq(k)*

purpose: inserts k as last queue element
exceptions: $\text{length}\geq m$
effects:

1) length='length'+1
2) if 'length'=0 then first=k
3) el(length)=k

O-function: *deq*

purpose: removes first queue element
exceptions: $\text{length}\leq 0$
effects:

1) length='length'-1
2) first='el'(1)
3) $\forall j{:}1\leq j\leq \text{length}$: el(j)='el'(j+1)

V-function names in single quotes refer to the previous value.

A problem with this specification technique arises from the fact that O-functions are described purely in terms of resulting changes to V-functions. This introduces a "delay" effect, in that the deq function above has no way to define the resulting value of first, purely in terms of 'first' and 'length'. This forces the introduction of the <u>hidden</u> V-function, "el", which is not callable from outside the queue module (c.f. the "enq2" operation of the algebraic specifications, with all the same criticisms).

In order to verify an implementation for an abstract state model specification, it is necessary to define "mapping V-functions" which map the abstract state onto the program state. For example, we might have

map(length): l (a program variable)
map(first): *head*↑ (dereference the pointer to the head element)
map(el(j)): *f(head,j)* where f(x,y)=if y=0 then x↑ else f(x↑.next,y-1)

for a linked list implementation of the queue. Then the verification of a particular function F with exceptions Φ and effect Ψ, consists of proving

$$\text{map}(\Phi)\ \{\text{program F}\}\ \text{map}(\Psi)$$

The notation is that of Hoare [Hoare 69] and is read "if map(Φ) is true and program F is invoked, then map(Ψ) will be true when it terminates."

This kind of specification has a more manageable verification method, since the proof of expressions like that above is well understood [Hoare 69, Floyd 67]. We still find that the lack of an explicit representation for the type of object in question is a problem, as it is with algebraic specifications, forcing the designer to invent "hidden" operators whose only purpose is to take its place.

2.2.3 Predicate Transformations

Both of the above specification techniques are representation-independent. That is, the specifications make no reference to particular data structures, thus leaving that decision to the implementor. We have seen that in many cases the purely functional approach forces the introduction of "hidden" operators, and we asserted that these hidden operators are merely a way around making use of actual structures. This is particularly evident from the abstract state machine model example, where the hidden V-function "el" strongly suggests the use of an array. Of course, the implementor is by no means forced to use an array - that just happens to be a convenient "abstract representation."

We feel comfortable with the use of abstract representations in the specification of abstract data types, because of our confidence in the combined data/control nature of the data type mechanism itself. Since no purely data-oriented nor control-oriented mechanism is sufficient, there is no reason to believe that either should be sufficient as a specification technique.

If the specification of an abstract data type includes a mathematical representation, then the specification of the operators can be given in terms of their effect on that representation, rather than in terms of their effect on each other. That effect can be stated in terms of predicate transformation in the same way that the effect of primitive programming language features is described. The proof procedure given here is largely based on the work of Hoare [Hoare 72b] for the verification of Simula classes, and the subsequent development by the Alphard group at CMU [Wulf 76]. We shall, however, use weakest pre-condition semantics [Dijkstra 76] for our specifications and verification instead of the weak correctness[1] method of Hoare [Hoare 69]. Appendix A contains the necessary definitions. The reader who is not entirely familiar with this approach should read the Appendix before continuing.

Every data type has both an abstract (mathematical) and a concrete (programming language) representation or structure. Additionally, each operator has both an abstract and concrete implementation. The task of verifying the correctness of an implementation consists of showing that it is a valid model for the specifications. To accomplish this, it is necessary to define a mapping, $\mathscr{A}$, from the concrete representation to the abstract one.

Axioms for operations take the form

$$P(X) \; \{ \; F(X) \; \} \; Q(X)$$

which is interpreted as "If P(X) holds and F(X) is executed, Q(X) will hold whenever F(X) terminates." Then, to verify this axiom for the abstract object X and program F, against the concrete object x and program f, we must prove

$$P(\mathscr{A}(x)) \Rightarrow wp(f(x), Q(\mathscr{A}(x))$$

which, by the nature of wp, guarantees that the operation will terminate. To carry out this proof, it is necessary that the function $\mathscr{A}$ be well-defined both before and after the execution of each concrete operation - otherwise the verification is meaningless. If we characterize the domain of $\mathscr{A}$ as the set of concrete objects which satisfy $\mathscr{J}$, i.e.

$$\text{domain}(\mathscr{A}) = \{c \mid \mathscr{J}(c)\}$$

then if we can show the invariance of $\mathscr{J}$ across each operator, we can assert that $\mathscr{A}$ is well-defined. The method by which this invariance is proven is presented in [Wegbreit 76] under the name of Generator Induction, and was indicated informally in [Hoare 72b]. Let the operators of type t be o_i, $i \in 1..n$, and let the set of objects of type t be T. Furthermore, let P

[1] *Weak* correctness is input-output correctness without termination. *Strong* correctness is termination only. *Partial* correctness is equivalent to weak correctness. *Total* correctness includes both weak and strong correctness.

be $(\forall x \in T)\ \mathcal{J}(x)$. Then if P is preserved by each o_i, $\mathcal{J}$ is invariant for t. Note that this formulation allows for the creation of new objects by any operator, as long as they are initialized to satisfy $\mathcal{J}$. It is assumed that this is the only way to create new objects.

2.2.4 The Character of Abstract Predicates

We indicated in section 2.1.3 that new data types are built up out of existing ones. This is done by applying structuring mechanisms to those types. These structuring mechanisms consist primarily of arrays, records, and references [Flon 74]. Since we are to define mathematical representations for the abstract types we specify, these are only naturally created by applying mathematical structuring mechanisms to other abstract types. Because we have not restricted ourselves to any particular syntactic form, we are free, in principle, to use any mathematical notions which we find appropriate. This somewhat disconcerting thought - disconcerting because we might all be required to be mathematicians and programmers in order to read or construct specifications - is fortunately not so terrible after all. It appears that a small number of concepts suffices for most purposes. These include the notions of set, sequence, vector, and cartesian product. Taking the axioms of set theory as given, we can define the other concepts axiomatically as is done in [Hoare 72a]. A series of definitions is given in Appendix B, and should be scanned before continuing further.

As an example, consider the bounded queue discussed previously:

Abstract representation: sequence(t)

Let S:sequence(t),
m:integer, <the maximum queue length>
x,k:t,
q:queue

Operator axioms:

$$\text{length}(S)<m \wedge q=S\ \{\ \text{enq}(q,k)\ \}\ q=S\sim k$$

$$q=k\sim S\ \{\ x\leftarrow\text{deq}(q)\ \}\ x=k \wedge q=S$$

$$q=k\sim S\ \{\ x\leftarrow\text{first}(q)\ \}\ x=k \wedge q=k\sim S$$

We give an implementation for these specifications as:

```
type queue(t:type, m:integer) =

    var V:array [0..m-1] of t,
        head,tail:integer;

    op create(q:queue)=
        if m≤0 then error
        else
            head←0; tail←0
        fi;

    op enq(q:queue,k:t)=
        if q.tail-q.head≥m then error
        else
            q.V[q.tail mod m]←k;
            q.tail←q.tail+1
        fi;

    op deq(q:queue):t =
        if q.tail≤q.head then error
        else
            q.head←q.head+1;
            q.V[(q.head-1) mod m]
        fi;

    op first(q:queue):t = if q.tail≤q.head then error else q.V[q.head mod m] fi

endtype
```

We define $\mathcal{A}(q) = \mathrm{seq}(q.V, q.head, q.tail-1)$, where

$$\begin{aligned} \mathrm{seq}(V,i,j) &= V[i \bmod m] \sim V[(i+1) \bmod m] \sim \ldots V[j \bmod m], \qquad \text{if } i \leq j \\ &= \lambda \qquad \text{if } i>j. \end{aligned}$$

We will drop the prefix "q." in what follows. Note that

$$\begin{aligned} \mathrm{length}(\mathcal{A}(q)) &= \mathrm{length}(\mathrm{seq}(V,head,tail-1)) \\ &= (tail-1 \bmod m) - (head \bmod m) + 1 \\ &= tail-head \end{aligned}$$

Additionally, we require

$$\mathcal{J} = 0 \le \mathrm{tail}-\mathrm{head} \le m \wedge m>0 \wedge \mathrm{tail} \ge 0 \wedge \mathrm{head} \ge 0$$

The fact that $\mathcal{J}$ is invariant can be derived as follows:

(1) $\mathcal{J} \Rightarrow \mathrm{wp}(\mathrm{enq}(q,k), \mathcal{J})$

$\mathcal{J} \Rightarrow \mathrm{wp}(\mathrm{enq}(q,k), 0 \le \mathrm{tail}-\mathrm{head} \le m \wedge m>0 \wedge \mathrm{tail} \ge 0 \wedge \mathrm{head} \ge 0)$

$\mathcal{J} \Rightarrow \mathrm{tail}-\mathrm{head}<m \Rightarrow 0 \le \mathrm{tail}-\mathrm{head}+1 \le m \wedge m>0 \wedge \mathrm{tail} \ge -1 \wedge \mathrm{head} \ge 0$

true

(2) $\mathcal{J} \Rightarrow \mathrm{wp}(x \leftarrow \mathrm{deq}(q), \mathcal{J})$

$\mathcal{J} \Rightarrow \mathrm{wp}(x \leftarrow \mathrm{deq}(q), 0 \le \mathrm{tail}-\mathrm{head} \le m \wedge m>0 \wedge \mathrm{tail} \ge 0 \wedge \mathrm{head} \ge 0)$

$\mathcal{J} \Rightarrow \mathrm{tail}>\mathrm{head} \Rightarrow 1 \le \mathrm{tail}-\mathrm{head} \le m+1 \wedge m>0 \wedge \mathrm{tail} \ge 0 \wedge \mathrm{head} \ge -1$

true

(3) $\mathrm{true} \Rightarrow \mathrm{wp}(\mathrm{create}(q), \mathcal{J})$

$\mathrm{wp}(\mathrm{create}(q), 0 \le \mathrm{tail}-\mathrm{head} \le m \wedge m>0 \wedge \mathrm{tail} \ge 0 \wedge \mathrm{head} \ge 0)$

$m>0 \wedge 0 \le 0 \le m \wedge m>0 \wedge 0 \ge 0 \wedge 0 \ge 0$

true

It remains to verify the operator axioms. For enq, we must prove

$$\mathrm{length}(S)<m \wedge \mathcal{A}(q)=S \;\Rightarrow\; \mathrm{wp}(\mathrm{enq}(q,k), \mathcal{A}(q)=S\sim k)$$

The post-condition expands to

$$S\sim k = \mathrm{seq}(v,\mathrm{head},\mathrm{tail}-1)$$

Then,

$$\begin{aligned}
&\mathrm{wp}(\mathrm{enq}(q,k), S\sim k=\mathrm{seq}(V,\mathrm{head},\mathrm{tail}-1)) = \\
&\qquad \mathrm{tail}-\mathrm{head}<m \wedge S\sim k=\mathrm{seq}(<V,\mathrm{tail} \bmod m,k>,\mathrm{head},\mathrm{tail}) = \\
&\qquad \mathrm{tail}-\mathrm{head}<m \wedge S\sim k=\mathrm{seq}(V,\mathrm{head},\mathrm{tail}-1)\sim k = \\
&\qquad \mathrm{length}(S)<m \wedge S=\mathcal{A}(q)
\end{aligned}$$

Similarly, for deq we must prove

$$\mathcal{A}(q)=k\sim S \;\Rightarrow\; \mathrm{wp}(x\leftarrow \mathrm{deq}(q),\ \mathcal{A}(q)=S \land x=k)$$

The post-condition expands to

$$x=k \land S=\mathrm{seq}(V,\mathrm{head},\mathrm{tail}-1)$$

and then

$$\begin{aligned}
&\mathrm{wp}(x\leftarrow \mathrm{deq}(q),\ x=k \land S=\mathrm{seq}(V,\mathrm{head},\mathrm{tail}-1)) = \\
&\qquad \mathrm{tail}>\mathrm{head} \land V[\mathrm{head}\ \mathrm{mod}\ m]=k \land S=\mathrm{seq}(V,\mathrm{head}+1,\mathrm{tail}-1) = \\
&\qquad \mathrm{tail}-\mathrm{head}>0 \land k\sim S=\mathrm{seq}(V,\mathrm{head},\mathrm{tail}-1) = \\
&\qquad \mathrm{length}(q)>0 \land \mathcal{A}(q)=k\sim S = \\
&\qquad \mathcal{A}(q)=k\sim S
\end{aligned}$$

For first we must prove

$$\mathcal{A}(q)=k\sim S \;\Rightarrow\; \mathrm{wp}(x\leftarrow \mathrm{first}(q),\ \mathcal{A}(q)=k\sim S \land x=k)$$

The post-condition expands to

$$x=k \land k\sim S=\mathrm{seq}(V,\mathrm{head},\mathrm{tail}-1)$$

and then

$$\begin{aligned}
&\mathrm{wp}(x\leftarrow \mathrm{first}(q),\ x=k \land k\sim S=\mathrm{seq}(V,\mathrm{head},\mathrm{tail}-1)) = \\
&\qquad \mathrm{tail}>\mathrm{head} \land V[\mathrm{head}\ \mathrm{mod}\ m]=k \land k\sim S=\mathrm{seq}(V,\mathrm{head},\mathrm{tail}-1) = \\
&\qquad \mathcal{A}(q)=k\sim S
\end{aligned}$$

2.3. On the Correctness of Formal Specifications

Now that we have seen how we can specify a module (data type) and verify a given implementation, we come to the question of what we can say about the correctness of the specifications themselves.

The process of formal specification involves both a decision as to the abstract representation of an object and decisions as to the ways in which that representation may be perturbed (by the operations of the type). These perturbations are usually subject to constraints which are not fully described by the choice of representation. For example, a sequence of elements of type t may be constrained so that the ordering of its elements satisfies some property Ω, i.e.

type seq2 =

abstract representation: sequence(t) *such that* $(\forall S \in seq2)\ \Omega(S)$

This constraint on the abstract representation is highly analogous to the constraint imposed on a concrete representation by the invariant which describes the domain of $\mathcal{A}$, and it has been called the *abstract invariant* of a data type [Wulf 76]. What is missing from that treatment is a discussion of the relationship between the abstract invariant and the abstract specifications. While there is no way to guarantee that the specifications are "correct" with respect to the highly abstract model possessed by the human who constructed them, nevertheless we can go a long way by establishing that the operations maintain the consistency of the abstract representation. The problem then is, given an abstract invariant for type t, $\mathcal{I}_a$, and a set of formal specifications of the form

$$P_j\ \{op_j\}\ Q_j$$

to establish that each such specification satisfies

$$\mathcal{I}_a \wedge P_j\ \{op_j\}\ \mathcal{I}_a$$

Actually, we must be careful in our treatment of free variables in the various predicates. In particular, it is clear that the assertion

$$v=v_0$$

is *not* invariant across the operation specified by

$$v=v_0-1\ \{op\}\ v=v_0$$

and yet the conjunction $(v=v_0) \wedge (v=v_0-1)$ is false, and

$$\text{false}\ \{op\}\ Q$$

is always true. We solve this problem by restricting the free variables of $\mathcal{I}_a$. Let x be the list of variables free in P, Q which may have different values in Q than they have in P. These are the parameters of op. $\mathcal{I}_a$ must contain no free variables which are not contained in x. The task then is to derive

$$\mathcal{I}_a \wedge P\ \{op(x)\}\ \mathcal{I}_a$$

from

$$P \; \{op(x)\} \; Q$$

Using the Adaptation and Consequence rules of [Hoare 71b], we can specify the conditions which will allow this inference. Specifically, let k be the list of variables free in P, Q but not in x. Then from the rule of Adaptation:

$$\frac{\vdash P \; \{op(x)\} \; Q}{\vdash \exists k(P \wedge \forall x(Q \Rightarrow \mathscr{I}_a)) \; \{op(x)\} \; \mathscr{I}_a}$$

and the rule of Consequence:

$$\frac{\vdash P \; \{op(x)\} \; Q, \; \vdash R \Rightarrow P}{\vdash R \; \{op(x)\} \; Q}$$

we obtain the rule of <u>Invariance</u>:

$$\frac{\vdash P \; \{op(x)\} \; Q, \; \vdash \mathscr{I}_a \wedge P \Rightarrow \exists k(P \wedge \forall x(Q \Rightarrow \mathscr{I}_a))}{\vdash \mathscr{I}_a \wedge P \; \{op(x)\} \; \mathscr{I}_a}$$

As a brief example, consider the specification of a simple type for positive integers:

<u>type</u> posint =

abstract representation = integer <u>such</u> <u>that</u> $(\forall z \in \text{posint}) z>0$

<u>Operations</u>

Let q, r, s: posint.

1) $r=a \wedge s=b$ {plus(q,r,s)} $r=a \wedge s=b \wedge q=a+b$

2) $r=a \wedge s=b \wedge a>b$ {minus(q,r,s)} $r=a \wedge s=b \wedge q=a-b$

<u>endtype</u>

Here $\mathscr{I}_a$ is the predicate $(\forall z \in \text{posint}) z>0$. To verify that the specifications leave $\mathscr{I}_a$ invariant, we must prove

$$(\forall z\in \text{posint})z>0 \wedge r=a \wedge s=b \ \{\text{plus}(q,r,s)\}\ (\forall z\in \text{posint})z>0$$

and

$$(\forall z\in \text{posint})z>0 \wedge r=a \wedge s=b \wedge a>b \ \{\text{minus}(q,r,s)\}\ (\forall z\in \text{posint})z>0$$

From the rule of Invariance, we must therefore show

$$\begin{aligned}&(\forall z\in \text{posint})z>0 \wedge r=a \wedge s=b\\&\qquad\Rightarrow (\exists a,b)[r=a \wedge s=b \wedge\\&\qquad\qquad\qquad(\forall q,r,s\in \text{posint})[r=a \wedge s=b \wedge q=a+b \Rightarrow (\forall z\in \text{posint})z>0]]\end{aligned}$$

and

$$\begin{aligned}&(\forall z\in \text{posint})z>0 \wedge r=a \wedge s=b \wedge a>b\\&\qquad\Rightarrow (\exists a,b)[r=a \wedge s=b \wedge a>b \wedge\\&\qquad\qquad\qquad(\forall q,r,s\in \text{posint})[r=a \wedge s=b \wedge q=a-b \Rightarrow (\forall z\in \text{posint})z>0]]\end{aligned}$$

Both implications follow trivially.

3. Application

3.1. Introduction

The methodology of Chapter 2 is applicable to any large, properly decomposed program. As we discussed in Chapter 1, we are primarily interested in the verification of operating systems. In this chapter we shall present the design, specification, implementation, and verification of a low-level operating system module, the process dispatcher. A great deal of emphasis is placed upon verification of the design (specifications) before verification of the implementation. Before we attempt this task however, we must first consider some of the more global aspects of applying the methodology to operating systems. In the next few sections we examine the effect of structure and decomposition on the verifiability of operating systems. We also consider the important interaction between the language in which an operating system is wr'tten and the system itself.

3.2. On the Structure of An Operating System

The study of operating systems design was significantly influenced by three research efforts - the T.H.E. system of Dijkstra [Dijkstra 68], the RC4000 system of Brinch Hansen [Brinch Hansen 70], and the Multics system developed at MIT's project MAC [Organick 72]. The reason these systems have been so influential is the significant contribution each has made with regard to the structure of operating systems. The primary reason for the complexity of most commercial systems is not merely their huge size, but the fact that the complexity of a large-scale system is increased many times by the lack of a coherent, well-modularized structure. Since it is our goal to reduce the complexity of verifying operating systems, a good system structure is crucial.

The T.H.E. system (and later the Venus system [Liskov 72]) divided the various aspects of an operating system into several *layers* (or *levels*), which were arranged in a hierarchical manner. Each layer constituted a modification of the next lower one, and in this way the hardware was successively molded into a machine which was much more convenient to use. Each of the layers (above the definition of processes) was composed of a number of processes. Each process could ask a lower-level (but never a higher-level) process to do some work for it.

The RC4000 system introduced the *kernel* or *nucleus* approach to system structuring. This basic approach has since been used in several diverse systems, including the HYDRA multiprocessing system [Wulf 74]. The kernel of an operating system consists of the definition and management of primitive system features, including processes, memory management, and low-level I/O. This approach is intended to allow for the design of various outer shells, all using the same kernel, thus permitting the tailoring of systems for particular applications while maintaining a common basis. The kernel is typically the most protected part of the system, and the approach has been used to restrict the scope of verification of protection mechanisms [Schroeder 75]. The RC4000 system was designed for use with process control computers having diverse applications.

The Multics system introduced the idea of a highly modular and distributed system, with a protection structure that allows the dynamic replacement of system modules on a per-user basis. The distributed, non-hierarchical nature of the Multics system negatively affects its verifiability. In a non-hierarchical system, it becomes difficult to maintain the principle of maximal encapsulation, since each system module has the potential to call or be called by any other, introducing the possibility of indirect recursion.

Kernel systems are not fundamentally different from layered systems; it is simply that the kernel boundary has special properties. A layered structure has the advantage of restricting the scope of verification by eliminating cycles and recursion, so we shall want our operating system to be so constructed.

3.3. On Hierarchy

In [Parnas 74], Parnas points out that there are many possible hierarchical structures, and that any particular one is not defined until the parts (entities to be ordered) and the relation (that governs the hierarchy) are specified. The T.H.E. system was a hierarchy of abstract machines which consisted of processes, and the ordering relation was "gives work to." The Multics system is organized into modules, and there exists an ordering relation which is "more privileged than", although the hierarchy is not enforced on inter-module calls. The scheme we will use is most similar to that of the Family of Operating Systems (FAMOS) design project at CMU [Habermann 76]. The FAMOS system is organized in a layered manner, but the parts among which the hierarchy is enforced are subroutines, not modules nor processes. In that system, a module is allowed to be split among several levels, some of its functions residing at each level. The FAMOS design strategy may be more general than is necessary, and we hope we will not find it necessary to have modules cross level boundaries. This is chiefly because our modules (abstract data types) are small. The splitting of modules complicates the verification of invariants, since it becomes possible to transfer from one level of a module to a lower one via an arbitrary path, without first having placed the data structure in a consistent state.

3.4. On a Decomposition into Levels

Having decided upon the structuring technique, the decisions as to which functions belong at which levels becomes most important. A goal of the FAMOS system is that the several special purpose systems which are created for a particular machine have as much in common as possible. This is accomplished by designing the lower system levels so as to postpone decisions which are not sufficiently common to system family members. This leads to a rather pleasing logical structure. We present a hypothetical structure here which differs from that of the common levels of the FAMOS system primarily in the ordering of processes and M_p management[2]. It is:

level #	level function
9	user interface
8	file system
7	user peripherals
6	swapping
5	M_s I/O
4	M_s management
3	M_p management
2	synchronization
1	processes and P_c management
0	hardware

In the structure as shown, the criterion for placing one level above another is simply that the lower level has no need for the facilities of the higher one. Level 1 is responsible for the maintenance of a fixed number of processes and for the multiplexing of ready processes among the hardware processors. Above this level, the actual number of processors is unknown. Level 2 will define the synchronization mechanism to be used by processes (e.g. semaphores).

The next level, M_p management, is responsible for the allocation and deallocation of primary memory. This level is placed above that of synchronization because it may be that a

[2] The PMS notation of [Bell 71]: M_p (primary memory), M_s (secondary memory), P_c (central processor).

request for memory allocation cannot be satisfied at a particular time, and the requesting process must therefore be delayed until some allocated storage is freed.

A small digression is in order at this point. The FAMOS system assigns the responsibility for M_p management to a level below that of the definition of processes - in fact to the lowest software level. This was done because the lowest level, the one which allocates memory, also introduces the concept of "protected addressing environment", and it was felt that as much of the system as possible (i.e. everything but the lowest level) should be so protected. The memory overflow problem (i.e. running out of memory) is solved by using the general mechanism of the *software trap*, which is intended to model the behavior of the hardware trap facility. Using this mechanism, a level which encounters a condition which it is not prepared to handle can invoke a particular trap, allowing a higher, "smarter" level to fix the problem. It is claimed that an upward transfer of control, such as a trap, is not a violation of the hierarchy if there is no dependence, on the part of the trap initiator, on the successful result of that trap. While this appears to be true in principle, the fact is that the trap usually occurs in an "intermediate" state, requiring that control eventually return to the point at which the trap occurred in order to continue the original task. We (the author) feel that such dependent traps are to be avoided. Since we are not using the addressing environment scheme in any case, there is no reason to place memory management below processes and synchronization.

Returning to the explanation of our hypothetical system structure, level 4 is assigned the task of allocation and deallocation of secondary storage, e.g. disk, from tables kept in M_p. Level 5 is then in charge of transferring data between primary and secondary memory. The swapping system, at level 6, uses the facilities of levels 4 and 5 to multiplex the allocated processes in primary memory (essentially scheduling them in a much coarser fashion that that of level 1). Above that, level 7 will define access to the other peripherals in the system (e.g. line printer, terminals, magnetic tape, and card reader). Level 8 provides a file system for storing and retrieving temporary or permanent data from the peripherals. The user interface at level 9 then interacts with terminals to drive the rest of the operating system.

3.5. System Structure and Implementation Language

We sketched, in Chapter 2, a programming language that is strongly typed - i.e. every variable has a type, the type determines the legal operations which may be applied to that variable, and there are no implicit coercions between types. These attributes will greatly enhance the verifiability of programs. Unfortunately, we cannot define and implement that language, and then use that fixed language throughout system implementation, for the simple reason that the semantics of many language features are dependent upon the correct operation of the system itself. For example, it would not be proper to use a language with dynamic storage allocation facilities in order to implement any level below that of M_s management, and it would not be proper to use any built-in synchronization constructs at or below the level at which synchronization is defined.

We therefore adopt the notion that the implementation language is a changing entity. Each time a new level is implemented, that level affects the language that is to be used to implement the next level. The easiest and most common change that can occur is the simple introduction of new data types. Occasionally, the syntax and semantics of the language will change.

The base language, i.e. the one which we will use to implement processes, will be strongly typed, but contain no dynamic allocation facilities, no recursion, no process creation, and no synchronization constructs. The only data types in existence at this point are the primitive ones - *integer, boolean,* and *real.* When processes are implemented, we will add to the language the construct

$$\underline{\text{cobegin}}\ S_1 // S_2 // \ldots S_n\ \underline{\text{coend}}$$

which defines n processes, with process i executing statement S_i. The construct will be implemented by choosing n of the processes made available by level 1 and assigning them to the specified tasks.

Whatever synchronization mechanism is introduced by level 2, that mechanism will be available for the implementation of level 3. One possibility might be the conditional critical region [Brinch Hansen 73], another the monitor concept [Hoare 74], and a third might be path expressions [Flon 76]. Each will change the implementation language in its own way.

When the M_p management level is implemented, we will extend the language to offer dynamic storage allocation. Two operators, *new* and *free* are introduced [Flon 75]. These will be used to allocate and release an instance of a given data type, but only from inside the type definition. Creation of new objects from outside the type definition must be done via the *create* operation of that type, which can invoke *new* and *free.* This will allow us to easily verify proper initialization of all objects.

We imagine that level 4 (M_s management) will provide the language with a new data type, Msblock(size), to correspond to a block of secondary memory. Level 5 (M_s I/O) will provide operations to transfer Msblock's between M_s and M_p. Level 7, the file system, introduces the data type *file*, with its operations *open*, *close*, *read*, *write*, etc. Further extension of the language by the other levels is minimal.

We also emphasize that it is possible (necessary, in fact) to <u>remove</u> features and data types from the language as they become too primitive. This will enable us to reduce the scope of verifications. For example, the type Msblock will not be useful above the file system, and it would be dangerous to leave it around.

3.6. Design of the Process Level

Before presenting formal specifications for the process level of our system, let us consider the criteria we would like those specifications to satisfy. First, the process level is intended to hide from higher levels the actual number of hardware processors. This is accomplished by defining a process to represent the state of a particular computation. That is, it consists of a program, including global data and constants, and local data which, among other things, contain the necessary information for restarting that computation from the point at which it was last preempted. It is then possible to multiplex the processes among the real processors, switching among the different processes from time to time in order to give the appearance of continuous service to higher levels. A reasonable goal is that the processor multiplexing be done in a fair manner (we shall be more precise about the meaning of fair later on).

Since processes actually in execution have no useful task to perform from time to time, as when they are waiting for information from another source, we would like to separate the set of existing processes into those which can execute and those which are waiting. In fact, we will define three states for a process - *waiting*, *ready*, and *running*. A *waiting* process is not considered a candidate for assignment to a processor - the set of *ready* processes is the one from which such candidates are chosen. A process in actual execution on a processor is said to be *running*. The three states are mutually exclusive, and as our second goal we would like to guarantee that every process is in one of the three states, with corresponding implications, at all times. As an example of such implications, non-running processes must have their execution state saved. We will therefore restrict the transition to the *waiting* state to be made only by the operation *unready*, and that to the *ready* state only by the operation *ready*. In order that we may guarantee a degree of fairness in the scheduling, we shall not allow higher levels to choose <u>which</u> processes to run next (else the fairness proof could not be localized to the process level), although we will allow them to determine the scheduling points. For this we define the operation *preempt*. We defer the concept of "time-slice" to a higher level in order to avoid the quite separable problems of managing a hardware clock. We only require of *preempt* that it be called periodically. Determination of the particular period, or even if there should be (just) one, is something to be decided by subsequent performance evaluation.

Since we desire a somewhat realistic system, we will incorporate the notion of process priority in our design. With each process we associate a priority. A process will not be run when there are *ready* processes of higher priority. The priority of a given process is controlled by higher levels, but it must remain fixed while that process is *ready*.

3.7. Formal Specification of the Goals

In this section we present a set of specifications which satisfy the above goals. We strongly suggest that the reader be familiar with the notation and definitions of Appendix B before proceeding.

Let the abstract representation of a process be the record

(state:(*running*, *waiting*, *ready*), pcs:*Pcstate*, prty:*integer*, loaded:*boolean*)

where *Pcstate* is a machine dependent type whose representation is suitable for saving the shared execution state (e.g. program counter, relocation registers, general registers, etc.), and which provides the operations *unload* and *load* to save and restore that state.

Henceforth we will assume that there is only one hardware processor available, so that *load* and *unload* refer to that processor. Although we could have assumed a larger number of real processors, the resulting complexity would serve no pedagogic purpose. We also assume that our processor has two protection states, and that the system (at least this level) executes in the privileged one while processes do not. Specifically, *load* and *unload* have effect only upon the "user" environment.

Let the variable *current* refer to the currently running process (i.e. *current* is a reference[3] to type *process*). Then we desire that the operations at this level (*ready*, *unready*, and *preempt*) satisfy the invariant

$$\begin{aligned}\mathcal{I}_1(\text{current}) = (\forall p \in \text{process})\ [&(\text{p.state=ready} \Rightarrow \text{p.prty} \le \text{current}\!\uparrow\!\text{.prty}) \\ &\wedge (\text{p.state=running} \equiv (\text{p.loaded} \wedge \text{current=}\!\uparrow\!\text{p})] \\ \wedge\ &\text{current}\!\uparrow\!\text{.state=running}\end{aligned}$$

In order to satisfy this invariant, the decision as to which process to run next must be made by choosing a *ready* process of highest priority. In addition, in order to provide the fairness we discussed earlier, we will choose from the set of such candidates the one which has been *ready* the longest. Since the specification technique has no explicit notion of time, we must group the *ready* processes in an abstract data structure which makes that choice convenient. We choose a vector of sequences of processes for this purpose, with each element of the vector representing the processes at a given priority, and with the head of each sequence being the next process to run at that priority.

[3] We use the PASCAL notation: $r\uparrow$ means r dereferenced, $\uparrow t$ means a reference to the object t.

Let R[1..nprty] be the "ready-list" vector, with R[k] having type *sequence of process*. Then *ready*, *unready*, and *preempt* must also satisfy the invariant

$$\begin{aligned}\mathcal{J}_2(\text{R}) = (\forall \text{k} \in 1..\text{nprty})&\\ &[(\forall \text{p} \in \text{process})\ (\text{has}(\text{R}[\text{k}],\text{p}) \equiv (\text{p.prty=k} \wedge \text{p.state=ready}))\\ &\qquad \wedge\ (\#\{\text{p} \mid \text{has}(\text{R}[\text{k}],\text{p})\} = \text{length}(\text{R}[\text{k}]))]\end{aligned}$$

so that each sequence contains all *ready* processes of a given priority, and that no process appears more than once.

We can now attempt to properly specify the *ready*, *unready*, and *preempt* operations.

1) *ready*(p): Readying a process of lower priority than the current one.

pre[4]: $\text{p.state=waiting} \wedge \text{current}\uparrow\text{.prty} \geq \text{p.prty} \wedge \text{R[p.prty]=S}$

post: $\text{p=<p',state,ready>} \wedge \text{current=current'} \wedge \text{R[p.prty]=S}\sim\text{p}$

2) *ready*(p): Readying a process of higher priority than the current one.

pre: $\text{p.state=waiting} \wedge \text{current=}\uparrow\text{c} \wedge \text{c.prty<p.prty} \wedge \text{R[c.prty]=S}$

post: $\text{current=}\uparrow\text{p} \wedge \text{p=<<p',state,running>,loaded,true>}$
$\wedge\ \text{c=<<c',state,ready>,loaded,false>} \wedge \text{R[c.prty]=S}\sim\text{c}$

3) *unready*(p): Unreadying a process other than the current one.

pre: $\text{p.state=ready} \wedge \text{R[p.priy]=S}\sim\text{p}\sim\text{V}$

post: $\text{p=<p',state,waiting>} \wedge \text{R[p.prty]=S}\sim\text{V} \wedge \text{current=current'}$

[4] In the form discussed for specification in Chapter 2, this would read *pre* {ready(p)} *post*.

4) *unready*(p): Unreadying the current process.

pre: $\text{current}=\uparrow p \wedge k=(>j)(R[j]\neq\lambda) \wedge R[k]=c\sim S$

post: $p=\langle\langle p',\text{state},\text{waiting}\rangle,\text{loaded},\text{false}\rangle \wedge \text{current}=\uparrow c$
$\wedge\ c=\langle\langle c',\text{state},\text{running}\rangle,\text{loaded},\text{true}\rangle \wedge R[c.\text{prty}]=S$

5) *preempt*(): *Pre-empting with no ready process of equal priority to the current one.*

pre: $(>j)(R[j]\neq\lambda) < \text{current}\uparrow.\text{prty}$

post: $\text{current}=\text{current}'$

6) *preempt*(): Pre-empting while a process of equal priority to the current one is ready.

pre: $\text{current}=\uparrow c \wedge p.\text{prty}=c.\text{prty} \wedge R[p.\text{prty}]=p\sim S$

post: $\text{current}=\uparrow p \wedge p=\langle\langle p',\text{state},\text{running}\rangle,\text{loaded},\text{true}\rangle$
$\wedge\ c=\langle\langle c',\text{state},\text{ready}\rangle,\text{loaded},\text{false}\rangle \wedge R[p.\text{prty}]=S\sim c$

Note that each of the above specified process operations happens to have two axioms associated with it. The two axioms are simply a convenient form of expressing the fact that there are two basic choices as to the outcome of the operation, and strongly suggest an outer-level control structure of the form

$$\underline{\text{if}}\ P_1\ \underline{\text{then}}\ S_1\ \underline{\text{elsif}}\ P_2\ \underline{\text{then}}\ S_2\ \underline{\text{else}}\ \underline{\text{error}}\ \underline{\text{fi}}$$

where $P_1\ \{S_1\}\ Q_1$ and $P_2\ \{S_2\}\ Q_2$ correspond to the two axioms.

3.8. Verification of the Goals

3.8.1 Verification of the Consistency Invariants

Before even considering an *implementation* for the specifications of type process, we must verify that these specifications satisfy our goals. In particular, we shall prove that both $\mathcal{J}_1$ and $\mathcal{J}_2$ are invariant across each axiom. The method used to prove this invariance is given in section 2.3. That is, to show $\mathcal{J}$ is invariant over op(x) when

$$P\ \{op(x)\}\ Q$$

is known, we must prove

$$\mathcal{J} \wedge P \Rightarrow \exists k(P \wedge \forall x(Q \Rightarrow \mathcal{J}))$$

where x is the parameter list, and k is the list of variables free in P,Q but not in x. In the case where k is empty, we can simplify this to

$$\mathcal{J} \wedge P \Rightarrow \forall x(Q \Rightarrow \mathcal{J})$$

The proof follows:

1) *ready*(p): We must prove

$$\mathcal{J}_1(\text{current}) \wedge \mathcal{J}_2(R) \wedge \text{p.state=waiting} \wedge \text{current}\uparrow\text{.prty} \geq \text{p.prty} \wedge R[\text{p.prty}]=S$$

$$\Rightarrow (\forall q \in \text{process})\ (\forall T \in 1..\text{nprty of sequence of process})$$
$$[q=\langle p,\text{state},\text{ready}\rangle \wedge \text{current=current}' \wedge T[\text{q.prty}]=S\sim q$$

$$\Rightarrow \mathcal{J}_1(\text{current}) \wedge \mathcal{J}_2(T)]$$

In this case, as in all of those that follow, the truth of the above expression can be ascertained by straightforward simplification, which we omit.

2) *ready*(p): We must prove

$$\mathcal{J}_1(\text{current}) \wedge \mathcal{J}_2(\text{R}) \wedge \text{p.state=waiting} \wedge \text{current}=\uparrow\text{c} \wedge \text{c.prty<p.prty} \wedge \text{R[c.prty]=S}$$

$$\Rightarrow (\forall \text{q,r,cur} \in \text{process})\ (\forall \text{T} \in 1..\text{nprty of sequence of process})$$
$$[(\text{cur}=\uparrow\text{q} \wedge \text{q=<<p,state,running>,loaded,true>}$$
$$\wedge\ \text{r=<<c,state,ready>,loaded,false>} \wedge \text{T[r.prty]=S}\sim\text{r})$$

$$\Rightarrow \mathcal{J}_1(\text{cur}) \wedge \mathcal{J}_2(\text{T})]$$

3) *unready*(p): We must prove

$$\mathcal{J}_1(\text{current}) \wedge \mathcal{J}_2(\text{R}) \wedge \text{p.state=ready} \wedge \text{R[p.prty]=S}\sim\text{p}\sim\text{V}$$

$$\Rightarrow (\forall \text{q} \in \text{process})\ (\forall \text{T} \in 1..\text{nprty of sequence of process})$$
$$[\text{q=<p,state,waiting>} \wedge \text{T[q.prty]=S}\sim\text{V} \wedge \text{current=current'}$$

$$\Rightarrow \mathcal{J}_1(\text{current}) \wedge \mathcal{J}_2(\text{T})]$$

4) *unready*(p): We must prove

$$\mathcal{J}_1(\text{current}) \wedge \mathcal{J}_2(\text{R}) \wedge \text{current}=\uparrow\text{p} \wedge \text{k}=(>\text{j})(\text{R[j]}\neq\lambda) \wedge \text{R[k]=c}\sim\text{S}$$

$$\Rightarrow (\forall \text{q,r,cur} \in \text{process})\ (\forall \text{T} \in 1..\text{nprty of sequence of process})$$
$$[(\text{q=<<p,state,waiting>,loaded,false>} \wedge \text{cur}=\uparrow\text{r}$$
$$\wedge\ \text{r=<<c,state,running>,loaded,true>} \wedge \text{T[r.prty]=S})$$

$$\Rightarrow \mathcal{J}_1(\text{cur}) \wedge \mathcal{J}_2(\text{T})]$$

5) *preempt*(): We must prove

$$\mathcal{J}_1(\text{current}) \wedge \mathcal{J}_2(\text{R}) \wedge (>\text{j})(\text{R[j]}\neq\lambda) < \text{current}\uparrow\text{.prty}$$

$$\Rightarrow (\text{current=current'}$$

$$\Rightarrow \mathcal{J}_1(\text{current}) \wedge \mathcal{J}_2(\text{R}))$$

6) *preempt*(): We must prove

$$\mathcal{J}_1(\mathrm{current}) \wedge \mathcal{J}_2(R) \wedge \mathrm{current}{=}\uparrow c \wedge p.\mathrm{prty}{=}c.\mathrm{prty} \wedge R[p.\mathrm{prty}]{=}p{\sim}S$$

$$\begin{aligned} &\Rightarrow (\forall q,r,\mathrm{cur}\in \mathrm{process})\ (\forall T\in 1..\mathrm{nprty\ of\ sequence\ of\ process}) \\ &\qquad [(\mathrm{cur}{=}\uparrow q \wedge q{=}\langle\langle p,\mathrm{state},\mathrm{running}\rangle,\mathrm{loaded},\mathrm{true}\rangle \\ &\qquad\qquad \wedge\ r{=}\langle\langle c,\mathrm{state},\mathrm{ready}\rangle,\mathrm{loaded},\mathrm{false}\rangle \wedge T[q.\mathrm{prty}]{=}S{\sim}r) \\ &\qquad\qquad\qquad \Rightarrow \mathcal{J}_1(\mathrm{cur}) \wedge \mathcal{J}_2(T)] \end{aligned}$$

This completes the proof that both $\mathcal{J}_1$ and $\mathcal{J}_2$ are invariant across the operations of the process module.

3.8.2 Verification of the Fairness Property

We will now attempt to prove that any implementation of the specifications for type process satisfies the fair service goal. In particular, we shall prove that the ready processes of highest priority are executed in Round Robin fashion. Although there may be many alternative definitions which may have their own merits, this one is reasonable both from the context of system behavior we possess thus far, and for the pedagogic purpose of illustrating the verification technique. Karp and Luckham [Karp 76] have verified a fairness property for a particular implementation of a process dispatcher. The proof we shall present, dealing with the specifications only, applies to any arbitrary implementation.

The proof takes the form of induction on the history of calls upon the operations *ready*, *unready*, and *preempt*. We shall require the addition of an auxiliary variable to the specifications, for the purpose of recording the successive values of the pointer *current*. This variable is denoted H, and will have type *sequence of process*. It is initially equal to the null sequence. Whenever a specification of the form

$$\mathrm{current}{=}\uparrow p \wedge p{\neq}c\ \{\mathrm{op}\}\ \mathrm{current}{=}\uparrow c$$

is executed (i.e. whenever current is changed across op), we will replace it by

$$\mathrm{current}{=}\uparrow p \wedge p{\neq}c \wedge H{=}S\ \{\mathrm{op}\}\ \mathrm{current}{=}\uparrow c \wedge H{=}S{\sim}c$$

The statement to be proven can now be cast as:

Theorem

Let *a* and *b* be arbitrary, distinct ready processes of equal priority, and let there be no non-*waiting* processes of higher priority. Then, for any sequence of calls on *ready*, *unready*, and *preempt* which maintains this state, the resulting history of dispatching, H, when reduced by deletion to a sequence of just *a*'s and *b*'s, will be of the form $(ab)*(a \vee \lambda)$.

Formally, let $\rho(S)$ reduce S to only a's and b's, i.e.

$$\begin{aligned}
&\rho(a) = a\\
&\rho(b) = b\\
&\rho(c) = \lambda \qquad \text{if } c\neq a \text{ and } c\neq b\\
&\rho(S\sim x) = \rho(S)\sim\rho(x)
\end{aligned}$$

Then without loss of generality we wish to prove the relation

$$\mathcal{I}_{fair}(R,H) = \rho(H\sim R[a.prty]) \in (ab)*(a\vee\lambda)$$

invariant over any history which maintains the state

$$\begin{aligned}
&a.prty=b.prty \wedge a.state\neq waiting \wedge b.state\neq waiting\\
&\qquad \wedge\ current\uparrow.prty\leq a.prty \wedge a.prty=(>k)(R[k]\neq\lambda)
\end{aligned}$$

Proof:

1) *ready*(p): We must prove

$$\begin{aligned}
&\mathcal{I}_{fair}(R,H) \wedge \mathcal{I}_1(current) \wedge \mathcal{I}_2(R)\\
&\qquad \wedge\ p.state=waiting \wedge current\uparrow.prty\geq p.prty \wedge R[p.prty]=S\\
&\qquad \Rightarrow (\forall q\in process)\ (\forall T\in 1..nprty \text{ of sequence of process})\\
&\qquad\qquad [q=<p,state,ready> \wedge current=current' \wedge T[q.prty]=S\sim q \wedge H=H'\\
&\qquad\qquad\qquad \Rightarrow \mathcal{I}_1(current) \wedge \mathcal{I}_2(T) \wedge \rho(H\sim T[a.prty]) \in (ab)*(a\vee\lambda)]
\end{aligned}$$

We can assume $p\neq a$ and $p\neq b$ (because p.state=waiting).
Furthermore $p.prty\leq a.prty$, else afterwards there will be a higher priority non-waiting process.

Then $\rho(H\sim T[a.prty]) = \rho(H\sim R[a.prty])$.

2) *ready*(p):

Since p.prty>current↑.prty, the conditions of the theorem are violated, and we need not consider this case.

3) *unready*(p): We must prove

$$\mathcal{I}_{\text{fair}}(\text{R,H}) \wedge \mathcal{I}_1(\text{current}) \wedge \mathcal{I}_2(\text{R}) \wedge \text{p.state=ready} \wedge \text{R[p.prty]=S}\sim\text{p}\sim\text{V}$$

$$\Rightarrow (\forall \text{q} \in \text{process})\ (\forall \text{T} \in \text{1..nprty of sequence of process})$$

$$[\text{q=<p,state,waiting>} \wedge \text{T[q.prty]=S}\sim\text{V} \wedge \text{current=current'} \wedge \text{H=H'}$$

$$\Rightarrow \mathcal{I}_1(\text{current}) \wedge \mathcal{I}_2(\text{T}) \wedge \rho(\text{H}\sim\text{T[a.prty]}) \in (\text{ab})*(\text{a} \vee \lambda)]$$

We can assume that $p \neq a$ and $p \neq b$, so $\rho(\text{H}\sim\text{T[a.prty]}) = \rho(\text{H}\sim\text{R[a.prty]})$.

4) *unready*(p): We must prove

$$\mathcal{I}_{fair}(R,H) \wedge \mathcal{I}_1(\text{current}) \wedge \mathcal{I}_2(R) \wedge \text{current}=\uparrow p \wedge k=(>j)(R[j]\neq\lambda) \wedge R[k]=c\sim S \wedge H=V$$

$$\Rightarrow (\forall q,r,cur\in \text{process})$$

$$(\forall T\in 1..nprty \text{ of sequence of process}) (\forall W\in \text{sequence of process})$$

$$[q=\langle\langle p,state,waiting\rangle,loaded,false\rangle \wedge cur=\uparrow r$$

$$\wedge\ r=\langle\langle c,state,running\rangle,loaded,true\rangle \wedge T[r.prty]=S \wedge W=V\sim r$$

$$\Rightarrow \mathcal{I}_1(cur) \wedge \mathcal{I}_2(T) \wedge \rho(W\sim T[a.prty]) \in (ab)*(a\vee\lambda)]$$

We assume $p\neq a \wedge p\neq b$.

i) Suppose $c=a$. Then $\rho(W)=\rho(V\sim a)=\rho(V)\sim a$ and $\rho(R[a.prty])=a\sim\rho(S)=a\sim\rho(T[a.prty])$.
So $\rho(W\sim T[a.prty])=\rho(V)\sim\rho(R[a.prty])=\rho(H\sim R[a.prty])$.

ii) If $c=b$, the symmetric argument holds.

iii) If $c\neq a \wedge c\neq b$, then $\rho(W\sim T[a.prty])=\rho(W)\sim\rho(T[a.prty])$
$=\rho(V)\sim\rho(R[a.prty])=\rho(H\sim R[a.prty])$.

5) *preempt*(): This is trivial.

6) *preempt*(): We must prove

$$
\begin{aligned}
&\mathcal{I}_{fair}(R,H) \wedge \mathcal{I}_1(current) \wedge \mathcal{I}_2(R)\\
&\quad \wedge current=\uparrow c \wedge p.prty=c.prty \wedge R[p.prty]=p\sim S \wedge H=V\\
&\quad \Rightarrow (\forall q,r,cur\in process)\ (\forall T\in 1..nprty \text{ of sequence of process})\\
&\qquad (\forall W\in \text{sequence of process})\\
&\qquad [cur=\uparrow q \wedge q=<<p,state,running>,loaded,true>\\
&\qquad\ \wedge r=<<c,state,ready>,loaded,false> \wedge T[q.prty]=S\sim r \wedge W=V\sim q\\
&\qquad\qquad \Rightarrow \mathcal{I}_1(cur) \wedge \mathcal{I}_2(T) \wedge \rho(W\sim T[a.prty]) \in (ab)*(a\vee\lambda)]
\end{aligned}
$$

i) Suppose $p=a$. Then $\rho(H)\in(ab)*$ and $\rho(S)=b$ or $\rho(S)=\lambda$.
So $\rho(W\sim T[a.prty])=\rho(H)\sim a\sim\rho(S)\sim\rho(r)$.
If $c=b$ then $\rho(S)=\lambda$ and $\rho(W\sim T[a.prty])=\rho(H)\sim a\sim b$.
If $c\neq b$ then $\rho(S)=b$ and $\rho(W\sim T[a.prty])=\rho(H)\sim a\sim b$.

ii) The symmetric argument holds for $p=b$.

iii) Assume $p\neq a \wedge p\neq b$. Then $\rho(W)=\rho(H)$.
If $c=a$ then $\rho(S)=b$ and $\rho(H)\in(ab)*a$, so $\rho(W\sim T[a.prty]) \in (ab)*aba$.
If $c=b$ then $\rho(S)=a$ and $\rho(H)\in(ab)*$, so $\rho(W\sim T[a.prty]) \in (ab)*ab$.

So $\mathcal{I}_{fair}$ is indeed invariant for any implementation of the given specifications. Thus, we have succeeded in verifying something which, though conceptually relatively simple, has not been rigorously proven in the past.

3.9. Implementation of the Process Level

The specifications for type process given in the previous section rely on the manipulation of the abstract data structure, R, which we called the ready-list (even though it isn't a list). Therefore, the programs for type process which implement those specifications must either implement a concrete ready-list modelled after R, or else rely upon a separate implementation of such a structure. For the moment we will assume the latter course.

Based upon the manipulation of R in the specifications, we postulate a new type, multiq, of which the ready-list is an instance. Its specifications are:

Abstract Structure:

$$\mathit{multiq}(\mathrm{t}{:}\mathit{type},\mathrm{nl}{:}\mathit{integer}) = \mathit{vector}\ 1..\mathrm{nl}\ \mathit{of\ sequence\ of}\ \mathrm{t}$$

Operations:

1) append(M:*multiq*,n:1..nl,x:*t*): Appends x to M[n].

$$\mathrm{wp}(\mathrm{append}(M,n,x),\ M[n]=S\sim k \wedge x=k) =$$

$$M[n]=S \wedge x=k \wedge (\forall j \in 1..\mathrm{nl})\neg \mathrm{has}(M[j],k)$$

2) delete(M:*multiq*,n:1..nl,x:*t*): Deletes x from M[n].

$$\mathrm{wp}(\mathrm{delete}(M,x),\ M[n]=V) =$$

$$V=S\sim T \wedge M[n]=S\sim x\sim T$$

3) j←highest(M:*multiq*): Returns the highest index in M of a non-null element.

$$\mathrm{wp}(j \leftarrow \mathrm{highest}(M),\ j=(>n)(M[n]\neq\lambda)) =$$

$$(\exists n)(M[n]\neq\lambda)$$

4) remove(M:*multiq*,n:1..nl): Returns the first element of M[n] after removing it.

$$wp(q \leftarrow remove(M,n),\ M[n]=S \wedge q=k) =$$

$$M[n]=k \sim S$$

Assuming these specifications, we can give the following straightforward implementation of type process:

```
type process =
      var pin:1..nproc;

      own pvec: array [1..nproc] of
                    record
                          state: (running,waiting,ready),
                          pcs: Pcstate,
                          loaded: boolean,
                          prty: 1..nprty
                    end,
            R: multiq(1..nproc,nprty),
            cur: 1..nproc;

      macro proc[p] = pvec[p.pin],
            cproc = pvec[cur];

      op ready(p:process) =
            if proc[p].state≠waiting then error
            elsif cproc.prty≥proc[p].prty then
                  proc[p].state←ready;
                  append(R,proc[p].prty,p.pin)
            else
                  unload(cproc.pcs); cproc.loaded←false;
                  cproc.state←ready;
                  append(R,cproc.prty,cur);
                  cur←p.pin;
                  cproc.state←running;
                  load(cproc.pcs); cproc.loaded←true
            fi;
```

```
op unready(p:process) =
        if proc[p].state=waiting then error
        elsif proc[p].state=ready then
                proc[p].state←waiting;
                delete(R,proc[p].prty,p.pin)
        else
                unload(cproc.pcs); cproc.loaded←false;
                cproc.state←waiting;
                cur←remove(R,highest(R));
                cproc.state←running;
                load(cproc.pcs); cproc.loaded←true
        fi;


op preempt =
        if highest(R)≥cproc.prty then
                begin
                var p:1..nproc;

                p←remove(R,cproc.prty);
                unload(cproc.pcs); cproc.loaded←false;
                cproc.state←ready;
                append(R,cproc.prty,cur);
                cur←p;
                cproc.state←running;
                load(cproc.pcs); cproc.loaded←true
                end
        fi

endtype
```

3.10. Verification of Type Process

The verification of the implementation just presented follows. We define

$\mathcal{A}(\mathrm{cur}) = \mathrm{current}$
$\mathcal{A}(\mathrm{cproc}) = \mathrm{current}\uparrow$
$\mathcal{A}(\mathrm{p{:}1..nproc}) = \mathrm{p{:}\uparrow process}$
$\mathcal{A}(\mathrm{proc[p]}) = \mathrm{p{:}process}$

maintaining the abstractions of Pcstate, (running,ready,waiting), and R. For this case, the well-definedness of $\mathscr{A}$ is guaranteed by static type checking. That is, the fact that the program compiles guarantees that it will always be in a state which is in the domain of $\mathscr{A}$. We now give the proof of each axiom of the specifications in turn:

1) *ready*(p): We must prove

proc[p].state=waiting ∧ cproc.prty≥proc[p].prty ∧ R[proc[p].prty]=S

⇒ wp(*ready(p)*, proc[p]=<proc'[p],state,ready> ∧ cur=cur' ∧ R[proc[p].prty]=S~p)

Computing the weakest-pre-condition, we obtain:

```
op ready(p:process) =
{ proc[p].state=waiting ∧ cproc.prty≥proc[p].prty ⇒ cur=cur' ∧ R[proc[p].prty]=S
        ∧ (∀j∈1..nprty)¬has(R[j],p) }
    if proc[p].state≠waiting then error
    elsif cproc.prty≥proc[p].prty then
        { cur=cur' ∧ R[proc[p].prty]=S ∧ (∀j∈1..nprty)¬has(R[j],p) }
        proc[p].state←ready;
        { proc[p]=<proc'[p],state,ready> ∧ cur=cur' ∧ R[proc[p].prty]=S
            ∧ (∀j∈1..nprty)¬has(R[j],p) }
        append(R,proc[p].prty,p.pin);
    else . . .
    fi;
{ proc[p]=<proc'[p],state,ready> ∧ cur=cur' ∧ R[proc[p].prty]=S~p }
```

The weakest pre-condition is implied by the given pre-condition and the invariant $\mathscr{I}_2$(R).

2) *ready*(p): We must prove

```
proc[p].state=waiting ∧ cur=c ∧ proc[c].prty<proc[p].prty ∧ R[proc[c].prty]=S

⇒ wp(ready(p), cur=p ∧ proc[p]=<<proc'[p],state,running>,loaded,true>
        ∧ proc[c]=<<proc'[c],state,ready>,loaded,false> ∧ R[proc[c].prty]=S~c)
```

Computing the weakest-pre-condition, we obtain:

```
op ready(p:process) =
{ proc[p].state=waiting ∧ cproc.prty<proc[p].prty ∧ cur=c ⇒ R[proc[c].prty]=S ∧ (∀j∈1..nprty)¬has(R[j],c) }
        if proc[p].state≠waiting then error
        elsif cproc.prty≥proc[p].prty then . . .
        else
                { cur=c ⇒ R[proc[c].prty]=S ∧ (∀j∈1..nprty)¬has(R[j],c) }
                unload(cproc.pcs); cproc.loaded←false; cproc.state←ready;
                { cur=c ⇒ proc[c]=<<proc'[c],state,ready>,loaded,false> ∧ R[proc[c].prty]=S
                        ∧ (∀j∈1..nprty)¬has(R[j],c) }
                append(R,cproc.prty,cur); cur←p.pin;
                { cur=p ∧ proc[c]=<<proc'[c],state,ready>,loaded,false> ∧ R[proc[c].prty]=S~c }
                cproc.state←running; load(cproc.pcs); cproc.loaded←true;
        fi;
{ cur=p ∧ proc[p]=<<proc'[p].state,running>,loaded,true>
        ∧ proc[c]=<<proc'[c],state,ready>,loaded,false> ∧ R[proc[c].prty]=S~c }
```

The weakest pre-condition is implied by the given pre-condition and $\mathcal{I}_2$(R).

3) *unready*(p): We must prove

$$\text{proc[p].state=ready} \wedge \text{R[proc[p].prty]=S}\sim\text{p}\sim\text{V}$$

$$\Rightarrow \text{wp}(\textit{unready(p)}, \text{proc[p]=<proc'[p],state,waiting>} \\ \wedge \text{R[proc[p].prty]=S}\sim\text{V} \wedge \text{cur=cur'})$$

Computing the weakest-pre-condition, we obtain:

```
op unready(p:process) =
{ proc[p] state=ready ⇒ cur=cur' ∧ R[proc[p] prty]=S~p~V }
        if proc[p].state=waiting then error
        elsif proc[p].state=ready then
                { cur=cur' ∧ R[proc[p] prty]=S~p~V }
                proc[p].state←waiting;
                { proc[p]=<proc'[p],state,waiting> ∧ cur=cur' ∧ R[proc[p] prty]=S~p~V }
                delete(R,proc[p].prty,p.pin)
        else . . .
        fi;
{ proc[p]=<proc'[p],state,waiting> ∧ R[proc[p] prty]=S~V ∧ cur=cur' }
```

The implication of the weakest pre-condition is clear.

4) *unready*(p): We must prove

$$cur=p \land k=(>j)(R[j]\neq\lambda) \land R[k]=c\sim S$$

$$\Rightarrow wp(\mathit{unready(p)},\ proc[p]=<<proc'[p],state,waiting>,loaded,false> \land cur=c$$
$$\land\ proc[c]=<<proc'[c],state,running>,loaded,true> \land R[proc[c].prty]=S)$$

Computing the weakest-pre-condition, we obtain:

```
op unready(p:process) =
{ proc[p].state≠waiting ∧ proc[p].state≠ready ∧ cur=p ⇒ proc[c].prty=(>j)(R[j]≠λ) ∧ R[proc[c].prty]=c~S }
        if proc[p].state=waiting then error
        elsif proc[p].state=ready then . . .
        else
                { cur=p ⇒ proc[c].prty=(>j)(R[j]≠λ) ∧ R[proc[c].prty]=c~S }
                unload(cproc.pcs); cproc.loaded←false; cproc.state←waiting;
                { proc[p]=<<proc'[p],state,waiting>,loaded,false>
                        ∧ proc[c].prty=(>j)(R[j]≠λ) ∧ R[proc[c].prty]=c~S }
                cur←remove(R,highest(R));
                { proc[p]=<<proc'[p],state,waiting>,loaded,false> ∧ cur=c ∧ R[proc[c].prty]=S }
                cproc.state←running; load(cproc.pcs); cproc.loaded←true;
        fi;
{ proc[p]=<<proc'[p],state,waiting>,loaded,false> ∧ cur=c
        ∧ proc[c]=<<proc'[c],state,running>,loaded,true> ∧ R[proc[c].prty]=S }
```

The implication of the weakest pre-condition is clear.

5) *preempt*(): We must prove

$$(>j)(R[j]\neq\lambda)<cproc.prty \Rightarrow wp(\mathit{preempt()},\ cur=cur')$$

This proof is trivial.

6) *preempt*(): We must prove

$$cur{=}c \wedge proc[p].prty{=}proc[c].prty \wedge R[proc[p].prty]{=}p{\sim}S$$

$$\Rightarrow wp(\mathit{preempt}(),\ cur{=}p \wedge proc[p]{=}{<}{<}proc'[p],state,running{>},loaded,true{>}$$
$$\wedge\ proc[c]{=}{<}{<}proc'[c],state,ready{>},loaded,false{>} \wedge R[proc[p].prty]{=}S{\sim}c)$$

Computing the weakest-pre-condition, we obtain:

```
op preempt =
{ (>j)(R[j]≠λ)≥cproc.prty ∧ cur=c ∧ cproc.prty=proc[p].prty ⇒
                R[proc[p].prty]=p~S ∧ (∀j∈1..nprty)¬has(R[j],c) }
        if highest(R)≥cproc.prty then
                begin
                var p:1..nproc;
                { cur=c ∧ cproc.prty=proc[p].prty ⇒ R[proc[p].prty]=p~S ∧ (∀j∈1..nprty)¬has(R[j],c) }
                p←remove(R,cproc.prty);
                { cur=c ⇒ R[proc[p].prty]=S ∧ (∀j∈1..nprty)¬has(R[j],c) }
                unload(cproc.pcs); cproc.loaded←false; cproc.state←ready;
                { cur=c ⇒ proc[c]=<<proc'[c],state,ready>,loaded,false> ∧ R[proc[p].prty]=S
                        ∧ (∀j∈1..nprty)¬has(R[j],c) }
                append(R,cproc.prty,cur);
                { proc[c]=<<proc'[c],state,ready>,loaded,false> ∧ R[proc[p].prty]=S~c }
                cur←p; cproc.state←running;
                load(cproc.pcs); cproc.loaded←true
                end
        fi;
{ cur=p ∧ proc[p]=<<proc'[p],state,running>,loaded,true>
        ∧ proc[c]=<<proc'[c],state,ready>,loaded,false> ∧ R[proc[p].prty]=S~c }
```

The weakest pre-condition is implied by the given pre-condition and $\mathcal{J}_2(R)$.

3.11. Implementation of Type Multiq

In the implementation and verification of type process, we relied upon the existence of another type, multiq, with which we implemented the ready-list. Now we come to the implementation of type multiq, and subsequently its verification. Refer to the specifications given in section 3.9.

The specifications clearly allow a completely separate implementation of type multiq, and a reasonable approach would be to construct such a type by using a vector of list pointers as the representation, i.e.

```
type multiq(t:type,nl:integer) =

        var H:array [1..nl] of el;
                .
                .
                .
endtype
```

where type *el* is (value:t, succ:1..nl).

While we could construct such an implementation without too much difficulty, it would be rather inefficient. Each process on the ready-list would require two pointers (one to the process and one to the next list element). Furthermore, each insertion or deletion would require storage allocation or de-allocation of the list elements.

If we had been constructing our system in a not-so-careful manner and without verification in mind, we would probably have implemented the ready-list as a "thread" through the processes themselves - i.e. we would extend the representation of a process to contain a pointer to other processes. With a little reflection we find, in fact, that we can still do just that. We re-declare the structure of type process to be:

```
type process =
        var pin:1..nproc;
        own pvec: array [1..nproc] of
                record
                        state: (running,ready,waiting),
                        pcs: Pcstate,
                        loaded: boolean,
                        prty: 1..nprty,
                        next: 1..nproc,
                        prev: 1..nproc
                end,
                R: array [1..nprty] of 0..nproc,
                cur: 1..nproc;
        .
        .
        .
endtype
```

Then we can see that the previous verification of type process is unaffected (because we deleted nothing and changed only the concrete type of R). Now we can implement the operations of type multiq as local procedures of type process, obtaining both a reasonable verification task for type multiq, and an efficient implementation thereof. Our implementation follows:

```
proc appendR(l:1..nprty,k:1..nproc) =
        if R[l]=0 then R[l]←k
        else
                swap(proc[k].succ, proc[proc[R[l]].pred].succ);
                swap(proc[k].pred, proc[R[l]].pred)
        fi;


proc removeR(l:1..nprty):1..nproc =
        begin
        var k: 1..nproc;
        k←R[l];
        if k=0 then error else deleteR(l,k) fi;
        k
        end;


proc deleteR(l:1..nprty,k:1..nproc) =
        if R[l]=k then
                if proc[k].succ=proc[k].pred
                        then R[l]←0
                        else R[l]←proc[k].succ
                fi
        else
                swap(proc[k].succ, proc[proc[k].pred].succ);
                swap(proc[k].pred, proc[proc[k].succ].pred)
        fi;


proc highestR:1..nprty =
        begin
        var k:1..nprty;
        k←nprty;
        while R[k]=0 do k←k-1 od;
        k
        end;
```

3.12. Verification of Type Multiq

The concrete invariant $\mathcal{J}$ of the given multiq implementation must fully capture the notion of a doubly likened circular list. The following is a concise definition of the concept:

$$\mathcal{J} = (\forall j \in 1..nproc)(j=proc[proc[j].succ].pred \wedge j=proc[proc[j].pred].succ)$$

That is, every process has the property that one step forwards (succ) followed by one step backwards (pred), or vice versa, will get you back to where you started from. Note that this must also be true of those processes which are not on a list (they must be self-referencing), and that is precisely what makes the link swapping implementation work.

Proving the above predicate invariant is a matter of applying the rules for array assignment and access[5] with appropriate simplification. We illustrate the method by showing $\mathcal{J}$ invariant across the appendR procedure.

Assume the concrete pre-condition to append R is

$$W = proc[k].succ=k \wedge proc[k].pred=k \wedge (\forall j \in 1..nprty)(R[j] \neq k)$$

That is, k is not a member of any list yet (because the only way to reach a self-referencing element is directly through the list headers in R, and this is guaranteed impossible). Compare this pre-condition with the abstract weakest pre-condition specified in section 3.9.

We must show

$$W \wedge \mathcal{J} \Rightarrow wp(appendR(l,k), \mathcal{J})$$

Clearly, if $R[l]=0$ initially then $R[l]=k$ will satisfy the invariant. The interesting case then is $R[l]>0$. Assume swap to have the definition

$$wp(swap(x,y), P) = P|^{y,x}_{x,y}$$

(simultaneous exchange of the variables x and y). Then

$$wp(swap(A[x],A[y]), A=A_0) = A=<<A_0, [x], A_0[y]>, [y], A_0[x]>$$

[5] See Appendix B.

Thus if R[l]>0, wp(appendR(l,k), $\mathcal{J}$ ∧ proc=P) =

```
      proc=
              < < < <P, [k].pred, P[R[l]].pred>,
                    [R[l]].pred, P[k].pred>,
                 [k].succ, P[P[R[l]].pred].succ>,
             [P[R[l]].pred].succ, P[k].succ>
```

We want to show

j=proc[proc[j].succ].pred ∧ j=proc[proc[j].pred].succ

Now proc[j].succ=

```
              if j=P[R[l]].pred then P[k].succ
              elsif j=k then P[P[R[l]].pred].succ
              else P[j].succ
```

Thus proc[proc[j].succ].pred =

```
              if proc[j].succ=R[l] then P[k].pred
              elsif proc[j].succ=k then P[R[l]].pred
              else P[proc[j].succ].pred
```

There are three possibilities for j:

1) Suppose j=P[R[l]].pred.

```
              Then proc[j].succ=P[k].succ
              and
                     proc[proc[j].succ].pred =
                            if P[k].succ=R[l] then P[k].pred
                            elsif P[k].succ=k then P[R[l]].pred
                            else P[P[k].succ].pred
```

From the pre-condition W, we know P[k].succ=k and R[l]≠k, so

proc[proc[j].succ].pred = P[R[l]].pred = j

2) Suppose $j=k \wedge j \neq P[R[l]].pred$.

```
Then proc[j].succ = P[P[R[l]].pred].succ
and
        proc[proc[j].succ].pred =
                if P[P[R[l]].pred].succ = R[l] then P[k].pred
                elsif P[P[R[l]].pred].succ = k then P[R[l]].pred
                else P[P[P[R[l]].pred].succ].pred
```

From $\mathcal{J}$, we know that P[P[R[l]].pred].succ = R[l], so

$$proc[proc[j].succ].pred = P[k].pred = k = j$$

3) Suppose $j \neq k \wedge j \neq P[R[l]].pred$.

```
Then proc[j].succ = P[j].succ
and
        proc[proc[j].succ].pred =
                if P[j].succ=R[l] then P[k].pred
                elsif P[j].succ=k then P[R[l]].pred
                else P[P[j].succ].pred
```

If P[j].succ=R[l] then P[P[j].succ].pred = j = P[R[l]].pred
which contradicts the hypotheses.
Furthermore, if P[j].succ=k then P[P[j].succ].pred = j = P[k].pred=k
which also contradicts the hypotheses. So

$$proc[proc[j].succ].pred = P[P[j].succ].pred = j$$

Similarly, we can establish

$$j=proc[proc[j].pred].succ$$

and we omit that proof. Using the same ideas we can show $\mathcal{J}$ to be invariant across all of the operations, and we omit those proofs also as not useful to the presentation.

Since $\mathcal{J}$ is invariant, the mapping from the concrete to the abstract, for which we shall use

$\mathscr{A}(R,proc) = R_a$ (the abstract object R used in the specifications)

$R_a[j]$ = if $R[j]=0$ then λ else seq(proc, R[j], R[j])

$$\begin{aligned}\text{seq(proc, x, y)} = \text{if } & \text{proc[x].succ=y then proc[x]}\\ & \text{else proc[x]} \sim \text{seq(proc, proc[x].succ, y)}\end{aligned}$$

is well-defined. To be rigorous we should have to prove that the invariant guarantees

$$(\exists n)\ \text{proc[R[l]].succ}^n = \text{R[l]}$$

(where $q.\text{succ}^n$ means q.succ.succ. . .succ n times) but we will take that as obvious. Under the above mapping, the pre-condition W becomes

$$\mathscr{A}(W) = (\forall j \in 1..\text{nprty}) \neg \text{has}(R_a[j],k)$$

Thus, for appendR, it only remains to show

$$\text{wp(appendR(l,k), } R_a[l]=S\sim k) = R_a[l]=S$$

From the definition of seq, we can deduce that

$$R_a[l]=S\sim k \equiv \text{proc[k].succ=first(S)} \wedge \text{proc[last(S)].succ=k}$$

Furthermore,

$$\text{first}(R_a[j]) = \mathscr{A}(\text{proc[R[j]]})$$

and

$$\text{last}(R_a[j]) = \mathscr{A}(\text{proc[proc[R[j]].pred]})$$

(assuming the interesting case when R[j]>0). Thus we must compute

$$\begin{aligned}&\text{wp(appendR(l,k),}\\ &\qquad \text{proc[k].succ=R[l]} \wedge \text{proc}[\text{proc}_0\text{[R[j]].pred].succ=k)}\end{aligned}$$

where proc_0 refers to the state of proc at entry. This works out to

$$\text{proc}_0\text{[k].succ=k} \wedge \text{proc}_0[\text{proc}_0\text{[R[l]].pred].succ=R[l]}$$

which, in light of the invariant, will map via $\mathscr{A}$ to

$$R_a[1] = S$$

While this proof has not been as rigorous as it might have been, it is convincing, and there is little to be gained from formally proving all of our assumptions (such as the obvious relationship between the *first* and *last* operators and $\mathscr{A}$. The proofs of removeR, deleteR, and highestR are straightforwardly similar and are not included.

4. Verifying General Parallel Programs

4.1. Introduction

The verification formalism for sequential programs is relatively well-understood, and indeed most of the proof rules for programs used in Chapter 2 are not new. In dealing with operating systems however, we are frequently faced with the problem of verifying parallel programs. A totally satisfactory formalism for this task has not yet appeared.

In this chapter we will explore two approaches to parallel program verification for a rather general syntactic class of programs. The first (section 4.2) uses Dijkstra's weakest pre-condition semantics [Dijkstra 76] to statically consider all of the possible execution paths of a system of parallel processes. The second (section 4.3) extends Owicki's methodology [Owicki 75] to handle arbitrary programs, without relying on a high-level synchronization mechanism. A methodology is given for proving loop termination in parallel programs.

4.2. Combinatoric Weakest Pre-Condition Semantics

4.2.1 Primitive Actions

It is a well-known fact that because a single high-level language statement is generally compiled into a sequence of machine instructions, it may be that a process is interrupted at a "mathematically inconvenient" place. This is reflected in the following classical example[6]:

$\{x=0\}$ ADD2: cobegin $x \leftarrow x+1$ // $x \leftarrow x+1$ coend $\{x=1 \lor x=2\}$

In spite of the fact that both processes appear to increment the variable x, if the incrementation is not indivisible the final value of x will be nondeterministic.

The difficulty in proving the correctness of such programs lies primarily in the fact that the primitive decomposition of "x←x+1" is not lexically explicit. In order to be able to prove such programs, it is necessary that the semantics of higher-level language statements

[6] As discussed in section 3.5, the *cobegin* . . . *coend* block defines the individual programs to be executed by each process.

include enough information to allow determination of the possible pre-emption points. For the case of simple incrementation, the definition

$$wp(x \leftarrow x+a,\ R) = R|^{x}_{x+a}$$

which suffices for the sequential case must be replaced by the definition

$$wp(x \leftarrow x+a,\ R) = wp(x_0 \leftarrow x;\ x_0 \leftarrow x_0+a;\ x \leftarrow x_0,\ R)$$

where x_0 is a unique temporary.

The fact that such definitions amount to explicitly specifying the way in which a given statement must be compiled is somewhat unfortunate since it removes much of the option of compiler writers. On the other hand, the program ADD2 can have entirely different behavior on the PDP-11 (where memory incrementation is a single instruction) than on the IBM 360-370 series (where the only way to increment memory is by loading, incrementing, and storing a register) if primitive semantics are not specified. If a concurrent programming language is to be supported on different machines and yet produce similar results for any given program, its semantic definition must be as precise as we have indicated.

4.2.2 Basic Semantics

In order to build up gradually to the general case, let us consider the semantics of parallel programs which consist of lexically explicit primitive actions and which consist only of sequences of assignment and when statements[7]. Then the semantics of such programs (for the two process case, with the obvious generalization) are expressed by

$$wp(\underline{\text{cobegin}}\ S_{11};\ S_{12};\ldots S_{1n}\ //\ S_{21};\ S_{22};\ldots S_{2n}\ \underline{\text{coend}},\ R) =$$

$$wp(\{<S_{11};\ S_{12};\ldots S_{1n}>,\ <S_{21};\ S_{22};\ldots S_{2n}>\},\ R)$$

where

[7] The statement *when* B *do* S *od* means "wait for B to become true, then execute S". The evaluation of B and the execution of S comprise one indivisible action.

1) wp({}, R) = R

2) wp({<>, <S>}, R) = wp({<S>, <>}, R) = wp({<S>}, R)

3) wp({<S>}, R) = wp(S, R)

4) wp({<S_{11}; S_{12}; . . .>, <S_{21}; S_{22}; . . .>}, R) =

   wp(choose 1 from {<S_{11}; S_{12}; . . .>, <S_{21}; S_{22}; . . .>}, R) $\wedge$
   wp(choose 2 from {<S_{11}; S_{12}; . . .>, <S_{21}; S_{22}; . . .>}, R)

5) wp(choose j from {<S_{11}; S_{12}; . . .>, <S_{21}; S_{22}; . . .>}, R) =

   if j=1 then wp(S_{11}; {<S_{12}; . . .>, <S_{21}; S_{22}; . . .>}, R)

   else if j=2 then wp(S_{21}; {<S_{11}; S_{12}; . . .>, <S_{22}; . . .>}, R)

6) wp(S; {P}, R) = wp(S, wp({P}, R))

The primitive actions of assignment and $\underline{\text{when}}$-$\underline{\text{do}}$ are defined as:

p1) wp(x←e, R) = $R|^{x}_{e}$

p2) wp($\underline{\text{when}}$ B $\underline{\text{do}}$ S $\underline{\text{od}}$, R) = B $\wedge$ wp(S, R)

As a brief example, consider the following computation:

```
wp(   cobegin
              P_1: when l=0 do l←1 od; T_1; l←0 //

              P_2: when l=0 do l←1 od; T_2; l←0
      coend
  , R) =
```

$$\mathrm{wp}(\{<\underline{\mathit{when}}\ l{=}0\ \underline{\mathit{do}}\ l\leftarrow 1\ \underline{\mathit{od}};\ T_1;\ l\leftarrow 0>,\ <\underline{\mathit{when}}\ l{=}0\ \underline{\mathit{do}}\ l\leftarrow 1\ \underline{\mathit{od}};\ T_2;\ l\leftarrow 0>\}, R) =$$

$$\mathrm{wp}(\underline{\mathit{when}}\ l{=}0\ \underline{\mathit{do}}\ l\leftarrow 1\ \underline{\mathit{od}};\ \{<T_1;\ l\leftarrow 0>,\ <\underline{\mathit{when}}\ l{=}0\ \underline{\mathit{do}}\ l\leftarrow 1\ \underline{\mathit{od}};\ T_2;\ l\leftarrow 0>\}, R) \wedge$$
$$\mathrm{wp}(\underline{\mathit{when}}\ l{=}0\ \underline{\mathit{do}}\ l\leftarrow 1\ \underline{\mathit{od}};\ \{<\underline{\mathit{when}}\ l{=}0\ \underline{\mathit{do}}\ l\leftarrow 1\ \underline{\mathit{od}};\ T_1;\ l\leftarrow 0>,\ <T_2;\ l\leftarrow 0>\}, R) =$$

. . . =

$$l{=}0 \wedge \mathrm{wp}(T_1, \mathrm{wp}(T_2, R)) \wedge \mathrm{wp}(T_2, \mathrm{wp}(T_1, R))$$

In performing the above computation, no simplification can be done until the expansion directed by rules 4 and 5 is finished. If, however, we choose to compute strongest post-conditions instead of weakest pre-conditions, we obtain some reduction in the complexity of the computation since simplification may be done along the way. For example, in the sequential case the weakest pre-condition proof of

$$\{\neg B\}\ \underline{\text{if}}\ B\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}\ \{R\}$$

involves the evaluation of

$$X = [B \Rightarrow wp(S_1, R)] \wedge [\neg B \Rightarrow wp(S_2, R)]$$

and the subsequent evaluation of $\neg B \Rightarrow X$. Using strongest post-condition directly requires the evaluation of

$$X = sp(S_2, \neg B)$$

and then $X \Rightarrow R$. The difference is primarily a reduction in the size of the predicate X.

For strongest post-condition, rule 4 becomes

4) $sp(\{<S_{11}; S_{12}; \ldots>, <S_{21}; S_{22}; \ldots>\}, R) =$

$$sp(\text{choose 1 from } \{<S_{11}; S_{12}; \ldots>, <S_{21}; S_{22}; \ldots>\}, R) \vee$$
$$sp(\text{choose 2 from } \{<S_{11}; S_{12}; \ldots>, <S_{21}; S_{22}; \ldots>\}, R)$$

with rules 1-3 and 5-6 having "wp" replaced by "sp". Rules p1 and p2 become

p1s) $sp(x \leftarrow e, R|^{x}_{e}) = R$

p2s) $sp(\underline{\text{when}}\ B\ \underline{\text{do}}\ S\ \underline{\text{od}}, R) = sp(S, B \wedge R)$

Although we regarded T_1 and T_2 as primitive in the previous exercise, in fact the same result is obtained when T_1 and T_2 are arbitrary sequences of assignment and when statements which are all free of access to the variable L.

If we express the ADD2 program of section 4.2.1 in terms of its primitive actions:

```
ADD2p:    cobegin
          a←x; a←a+1; x←a //
          b←x; b←b+1; x←b
          coend
```

we find

$$wp(\mathrm{Add2p},\ x=2) = \mathrm{false}$$

and

$$wp(\mathrm{Add2p},\ x=1 \vee x=2) = (x=0)$$

4.2.2.1 Conditionals

If we add to the language the statement

$$\underline{\mathrm{if}}\ B\ \underline{\mathrm{then}}\ T_1\ \underline{\mathrm{else}}\ T_2\ \underline{\mathrm{fi}}$$

then we must consider that even if B evaluates to true at some point, its value may change by the time T_1 is actually executed. To solve this problem we require that the primitive form of a conditional be such that B contains no divisible references to shared variables, and T_1 and T_2 are themselves in primitive form. For example, the statement

$$\underline{\mathrm{if}}\ c_1 \wedge c_2\ \underline{\mathrm{then}}\ x \leftarrow x+1\ \underline{\mathrm{else}}\ y \leftarrow y+1\ \underline{\mathrm{fi}}$$

must be converted to

```
$t_1 \leftarrow c_1$;

$t_1 \leftarrow t_1 \wedge c_2$;

if $t_1$ then $t_2 \leftarrow x$; $t_2 \leftarrow t_2+1$; $x \leftarrow t_2$

      else $t_3 \leftarrow y$; $t_3 \leftarrow t_3+1$; $y \leftarrow t_3$

fi
```

We can then modify rule 5 to be:

```
5') wp(choose j from $\{<S_{11}; S_{12}; \ldots>, <S_{21}; S_{22}; \ldots>\}$, R) =
    if j=1 then
        if $S_{11}$ is "if B then $T_1$ else $T_2$ fi"
            then  $[B \wedge wp(\{<T_1; S_{12}; \ldots>, <S_{21}; S_{22}; \ldots>\}, R)] \vee$
                  $[\neg B \wedge wp(\{<T_2; S_{12}; \ldots>, <S_{21}; S_{22}; \ldots>\}, R)]$
            else  $wp(S_{11}; \{<S_{12}; \ldots>, <S_{21}; S_{22}; \ldots>\}, R)$
    else if j=2 then
        .
        .
        .
```

4.2.2.2 Loops

To include while statements, i.e.

while B do T od

where T and B are in primitive form, we must consider that although the boolean B may evaluate to true and the loop may be entered, B may no longer hold by the time T is executed. We further change rule 5 to be:

```
5") wp(choose j from $\{<S_{11}; S_{12}; \ldots>, <S_{21}; S_{22}; \ldots>\}$, R) =
    if j=1 then
        if $S_{11}$ is "while B do T od" then
            $[B \wedge wp(\{<T;$ while B do T od; $S_{12}; \ldots>, <S_{21}; S_{22}; \ldots>\}, R)]$
            $\vee [\neg B \wedge wp(\{<S_{12}; \ldots>, <S_{21}; S_{22}; \ldots>\}, R)]$
        else if $S_{11}$ is "if B then $T_1$ else $T_2$ fi" then
            .
            .
            .
```

4.3. Extending the Axiomatic Approach

4.3.1 The Non-Interference Principle

Although we have presented on the previous pages a complete semantic characterization of parallel programs which consist of lexically explicit indivisible actions, the complexity of that characterization effectively prohibits its direct application to verification. Most of the difficulty arises from the fact that all possible execution sequences are individually treated, even if they fall into large equivalence classes based upon indistinguishable outcome. For example, the fact that

```
wp(  cobegin
          P1: while x>0 do x←x-1 od //
          P2: while y>0 do y←y-1 od
     coend
   ,true)
```

is equal to true is immediately obvious. However, the actual weakest pre-condition computation as defined must explore the hypothesis, "Suppose P_1 executes i cycles, then P_2 executes j cycles, then P_1 executes k cycles, etc.," for all i,j,k, etc. All such execution sequences are equivalent in the sense that there is nothing one process can do to affect the other in any way. This is the principle of *non-interference* introduced in [Owicki 75]. We give a precise definition in our own terms.

Let $\{P_j \mid j \in 1..n\}$ be the set of processes of a cobegin-coend block, and let each process j be represented by the program

$$S_j: \quad S_{j0}; S_{j1}; \ldots S_{jm}$$

where each S_{jk} is a separate statement. Let proof(Q_j, S_j, R_j) be any sequential proof of

$$Q_j \ \{ \ S_j \ \} \ R_j$$

i.e.

$$\{\alpha_{j0}\} \ S_{j0} \ \{\alpha_{j1}\} \ S_{j1} \ \{\alpha_{j2}\} \ldots S_{jm} \ \{\alpha_{j,m+1}\}$$

where $Q_j \Rightarrow \alpha_{j0}$, $\alpha_{j,m+1} \Rightarrow R_j$, and $\alpha_{jk} \Rightarrow \mathrm{wlp}(S_{jk}, \alpha_{j,k+1})$. Then the set

$$\{\ \mathrm{proof}(Q_j,\ S_j,\ R_j)\ |\ j\in 1..n\ \}$$

is a proof of the entire program if

$$(\forall j\in 1..n)\ (\forall k\in 0..m+1)\ (\forall i\neq j)\ (\forall r\in 0..m)\ [\alpha_{jk} \wedge \alpha_{ir} \Rightarrow \mathrm{wlp}(S_{ir},\ \alpha_{jk})]$$

That is, if no assertion α_{jk} of process j is interfered with (potentially invalidated) by any statement in any other process.

Note that if the interference-free property is not satisfied, that does not necessarily mean that the program is incorrect - only that the individual sequential proofs are not strong enough. For example, a sequential proof of

$$\{\mathrm{true}\}\ B\leftarrow\mathrm{true};\ A\leftarrow\mathrm{true}\ \{A\}$$

might be

$$\{\mathrm{true}\}\ B\leftarrow\mathrm{true};\ \{\mathrm{true}\}\ A\leftarrow\mathrm{true}\ \{A\}$$

or it might be

$$\{\mathrm{true}\}\ B\leftarrow\mathrm{true};\ \{B\}\ A\leftarrow\mathrm{true}\ \{A\}$$

The former will not confirm non-interference of the assertion {A⇒B} since

$$\{A\Rightarrow B \wedge \mathrm{true}\}\ A\leftarrow\mathrm{true}\ \{A\Rightarrow B\}$$

is not valid, while the latter will since

$$\{A\Rightarrow B \wedge B\}\ A\leftarrow\mathrm{true}\ \{A\Rightarrow B\}$$

is valid. Note that it is not necessary that a high-level synchronization mechanism be used to guarantee mutual exclusion. For example, the program

```
cobegin
    while x>0 do x←x-2 od //
    while x<0 do x←x+1 od
coend
```

has the proof

```
{true}
cobegin
        {true} while x>0 do {x>0} x←x-2 {x>-2} od {x=0 v x=-1} //
        {true} while x<0 do {x<0} x←x+1 {x<1} od {x=0}
coend
{x=0}
```

4.3.2 Proving Termination

4.3.2.1 Introduction

Owicki's work, which is based upon the non-interference property, deals primarily with weak correctness. We shall extend the utility of that work by showing how to prove loop termination for parallel programs. A corollary of being able to prove loop termination is, of course, being able to prove that a when statement will eventually be executed, since a when statement can be represented as a busy-wait loop.

4.3.2.2 The Sequential Case

We first review the method of proving sequential loop termination presented in [Dijkstra 76]. The goal is to find a predicate $\mathcal{I}$ which satisfies

$$\mathcal{I} \wedge \mathrm{wlp}(S, \mathcal{I}) \Rightarrow \mathrm{wp}(\underline{\mathrm{while}}\ B\ \underline{\mathrm{do}}\ S\ \underline{\mathrm{od}},\ \mathrm{true})$$

Let t be any bounded integer function of the program state, i.e. without loss of generality $t \geq 0$. Furthermore, suppose that

$$(1) \qquad \mathcal{I} \wedge B \Rightarrow t>0$$

and that S decreases t, i.e. if T is the program $t_0 \leftarrow t;\ S$, then

$$(2) \qquad \mathcal{I} \wedge B \Rightarrow \mathrm{wp}(T,\ t<t_0)$$

Then the loop must surely terminate, since (2) guarantees a steady decrease of t and (1) requires that when t reaches 0 (as it must), B will be false (because $\mathcal{I}$ is invariant). For example, the program

```
s,j←0;
while j≤N do s←s+A[j]; j←j+1 od
```

will terminate because $\mathcal{I} = (s = \sum_{k=0}^{j-1} A[k])$ is invariant, i.e.

$$s = \sum_{k=0}^{j-1} A[k] \wedge j \leq N \Rightarrow s + A[j] = \sum_{k=0}^{j} A[k]$$

(1) is guaranteed by $t = |N| - j + 1$, since

$$\mathcal{J} \wedge j \leq N \Rightarrow |N| - j + 1 > 0$$

(2) is guaranteed also, since

$$\mathcal{J} \wedge j \leq N \Rightarrow$$

$$wp(t_0 \leftarrow |N| - j + 1;\ s \leftarrow s + A[j];\ j \leftarrow j + 1,\ |N| - j + 1 < t_0)$$

$$\Rightarrow |N| - j < |N| - j + 1$$

$\mathcal{J}$ *is initially* satisfied since $\sum_{k=0}^{-1} A[k] = 0$ holds vaccuously, *and* $t \geq 0$ is initially satisfied since $|N| + 1 \geq 0$.

4.3.2.3 The Parallel Case

While the fact that this simple program terminates is rather obvious, we have presented a rigorous proof so that we may now observe the effect of other processes on the method. We have relied heavily on the fact that the loop body, S, decreases the termination function, t. This need not be true for parallel programs, for example

```
   x←N; y←0; B←true;
P: cobegin
          L1: while x>0 do S1: y←(y+1) mod N od; B←false //
          L2: while B do S2: x←x-1 od
   coend
```

At this point it is necessary to describe the behavior of the cobegin-coend block more precisely. In [Brinch Hansen 73], the program

$$\underline{\text{cobegin}}\ S_1\ //\ S_2\ //\ .\ .\ .S_n\ \underline{\text{coend}}$$

is said to behave as though the statements are executed *concurrently*, meaning that their executions may overlap in time. This definition is not really sufficient, because a uni-processor implementation may have radically different behavior from a multi-processor implementation. For example, suppose that the process scheduler of a uni-processor operating system does not timeslice at all. Then program P above will definitely not terminate, since there is never any reason to switch processes. Suppose, on the other hand, that the scheduler does timeslice. Then we can only say that P may terminate, since we have not prohibited priority scheduling in which, for example, L_1 may have higher priority than L_2.

However, if each process is assigned its own processor, program P is guaranteed to terminate, regardless of the relative speeds of the individual processors (as long as they are finite).

Since we must choose a suitable definition before we can verify programs, we will choose the multi-processor case. This decision is not at all arbitrary, since it reflects the current trend in hardware toward distributed processing power.

Returning to the analysis of program P, suppose we wish to prove termination of the loop L_1. Clearly the function governing termination is $t=pos(x)$, where

$$pos(x) = \text{if } x<0 \text{ then } 0 \text{ else } x$$

Unfortunately, S_1 does not affect t, although S_2 does. Let us consider sequential proofs of the two processes:

```
x←N; y←0; B←true;
{B}
cobegin
      L1:   {B}
            while x>0 do {B} S1: y←(y+1) mod N {B} od;
            {x≤0}
            B←false
            {x≤0 ∧ ¬B} //

      L2:   {true}
            while B do {true} S2: x←x-1 {true} od;
            {¬B}
coend
{x≤0 ∧ ¬B}
```

Both sequential proofs are interference-free. In process 1 only assertions about x are affected by process 2 and

$$x\le 0 \wedge \text{true} \Rightarrow wlp(x\leftarrow x-1,\ x\le 0)$$

In process 2 only assertions about B are affected by process 1 and

$$\neg B \wedge x\le 0 \Rightarrow wlp(B\leftarrow false,\ \neg B)$$

To prove termination of L_1, we must show that S_2, which decrements t=pos(x), is executed a sufficient number of times to reach t=0. This is clear because the non-termination of L_2, {B}, is invariant across S_1. Note that to prove termination of L_2, we must show that eventually "B←false" is executed, and we have that as a corollary to the termination of L_1. In light of this example we shall try to formalize a methodology for proving termination of parallel programs. First we introduce some important concepts.

We define a *weak loop invariant* to be any predicate which always holds during execution of a loop body, not just at entry. For example,

```
while true do
      {A ∧ B}
            B←false;
            {A}
            B←true
      {A ∧ B}
      od
```

Here only {A} is a weak invariant. (We call it *weak*, even though its invariance requirement is more stringent, because it must be a weaker predicate than the normal loop invariant.)

A *weak parallel loop invariant* is a weak loop invariant for which a proof of non-interference can be exhibited.

A *steady-state loop invariant* is any predicate which is guaranteed to be satisfied after a finite number of executions of the body, and which remains invariant thereafter. For example,

```
x←N; A←false;
while true do
      if x>0 then x←x-1 else A←true fi
      od
```

Here {A} is a steady-state loop invariant. Formally, for the loop while B do S od, P is a steady-state invariant iff at entry

$$[P \wedge B \Rightarrow wp(S, P)] \wedge (\exists k)G_k$$

where

$$G_0 = P$$

$$G_{k+1} = B \wedge wp(S, G_k)$$

That is, iff P is invariant and the loop

$$\underline{\text{while}}\ B \wedge \neg P\ \underline{\text{do}}\ S\ \underline{\text{od}}$$

terminates by establishing P.

A *steady-state weak loop invariant*, by analogy to the distinction between a normal and a weak invariant, is any steady-state loop invariant which is also a weak invariant.

Finally, a *steady-state weak parallel loop invariant* is a steady-state weak invariant for which a proof of non-interference can be exhibited.

To prove termination then, we proceed as follows:

1) Define an integer function, t, of the program state, and prove $t \geq 0$ is invariant.

2) Find a steady-state weak parallel invariant $\mathcal{J}$ for the loop in question, and prove

$$\mathcal{J} \wedge B \Rightarrow t>0$$

3) For those statements (in other processes and within the loop in question) which may affect the value of t, show that, in any state in which $\mathcal{J}$ holds, none of these can increase t unboundedly.

4) Show that in any state in which $\mathcal{J} \wedge B$ holds, some statements will cause a decrease in t.

5) Show that as long as $\mathcal{J} \wedge B$ holds, statements which decrease t must continue to be executed.

These conditions must imply termination of the loop in question, since a steady decrease in t and (2) imply termination when t=0.

4.3.3 An Example

We illustrate the method with the following example, which provides a fair solution to the mutual exclusion problem without the use of synchronization other than busy waiting. The problem was first discussed in [Dijkstra 65].

Two processes, A and B, each contain a critical section, and those critical sections must be prevented from executing simultaneously. Furthermore, should both processes desire execution at the same time, the decision as to which to grant permission to should be made on an alternating basis.

A solution is the following:

```
                var   inA, inB: boolean initially false,
                      prty: (A,B) initially A;

processA: while true do                         processB: while true do
   <think>                                         <think>
   inA←true;                                       inB←true;
   while inB do                                    while inA do
       if prty=B then                                  if prty=A then
           inA←false;                                      inB←false;
           while prty=B do nothing od;                     while prty=A do nothing od;
           inA←true                                        inB←true
           fi                                              fi
       od;                                             od;
   <critical section>                              <critical section>
   inA←false;                                      inB←false;
   prty←B                                          prty←A
   od                                              od
```

Weak correctness for this solution is established by the following proof, which is easily seen to be interference free. We include the auxiliary variables critA and critB in order to be able to state the mutual exclusion requirement.

```
var     inA, inB: boolean(false),
        prty: (A,B)(A);
aux var critA, critB: boolean(false);
```

```
processA: while true do
   {¬inA ∧ ¬critA ∧ critB⇒inB}
   <think>
<A0> inA←true;
   {inA ∧ ¬critA ∧ critB⇒inB}
<A1> while ¬(critA←¬inB) do
        {inA ∧ ¬critA ∧ critB⇒inB}
        if prty=B then
        <A2> inA←false;
             {¬inA ∧ ¬critA ∧ critB⇒inB}
        <A3> while prty=B do
                 {¬inA ∧ ¬critA ∧ critB⇒inB}
                 nothing
                 od;
             {¬inA ∧ ¬critA ∧ critB⇒inB ∧ prty=A}
        <A4> inA←true
             fi
   <A5> {inA ∧ ¬critA ∧ critB⇒inB ∧ prty=A}
        od;
   {inA ∧ critA ∧ ¬critB}
   <critical section>
   critA←false;
   {inA ∧ ¬critA ∧ ¬critB}
<A6> inA←false;
   {¬inA ∧ ¬critA ∧ critB⇒inB}
<A7> prty←B
   {¬inA ∧ ¬critA ∧ critB⇒inB}
   od
```

```
processB: while true do
   {¬inB ∧ ¬critB ∧ critA⇒inA}
   <think>
<B0> inB←true;
   {inB ∧ ¬critB ∧ critA⇒inA}
<B1> while ¬(critB←¬inA) do
        {inB ∧ ¬critB ∧ critA⇒inA}
        if prty=A then
        <B2> inB←false;
             {¬inB ∧ ¬critB ∧ critA⇒inA}
        <B3> while prty=A do
                 {¬inB ∧ ¬critB ∧ critA⇒inA}
                 nothing
                 od;
             {¬inB ∧ ¬critB ∧ critA⇒inA ∧ prty=B}
        <B4> inB←true
             fi
   <B5> {inB ∧ ¬critB ∧ critA⇒inA ∧ prty=B}
        od;
   {inB ∧ critB ∧ ¬critA}
   <critical section>
   critB←false;
   {inB ∧ ¬critB ∧ ¬critA}
<B6> inB←false;
   {¬inB ∧ ¬critB ∧ critA⇒inA}
<B7> prty←A
   {¬inB ∧ ¬critB ∧ critA⇒inA}
   od
```

Note that the expanded termination conditions of the original "while inA" and "while inB" loops (<B1> and <A1>) are quite legal on the grounds that we have simply expanded an indivisible action (the evaluation of inA or inB) into one which is guaranteed to terminate and which only changes variables which cannot affect the program's behavior.

4.3.3.1 Weak Correctness

The stated assertions follow immediately by taking successive post-conditions of the input assertions which appear at entry to the outer loops. Furthermore, each assertion is not interfered with by the other process. We present two illustrative examples of what must be proven to guarantee this.

The assertion

$$\{\mathrm{inA} \wedge \neg\mathrm{critA} \wedge \mathrm{critB}\Rightarrow\mathrm{inB} \wedge \mathrm{prty}=\mathrm{A}\}$$

at <A4> is not interfered with by the assignment at <B2> because

$$(\mathrm{inA} \wedge \neg\mathrm{critA} \wedge \mathrm{critB}\Rightarrow\mathrm{inB} \wedge \mathrm{prty}=\mathrm{A}) \wedge (\neg\mathrm{inB} \wedge \neg\mathrm{critB} \wedge \mathrm{critA}\Rightarrow\mathrm{inA})$$

$$\Rightarrow \mathrm{wlp}(\mathit{inB}\leftarrow\mathit{false},\ (\mathrm{inA} \wedge \neg\mathrm{critA} \wedge \mathrm{critB}\Rightarrow\mathrm{inB} \wedge \mathrm{prty}=\mathrm{A}))$$

Furthermore, the same assertion is not interfered with by the assignment at <B4> because its conjunction with the assertion at <B4> is false.

Note that the critical sections are guaranteed to be mutually exclusive (as long as they don't alter any of the variables inA, inB, and prty) because the assertions

$$\{\mathrm{inA} \wedge \mathrm{critA} \wedge \neg\mathrm{critB}\}$$

and

$$\{\mathrm{inB} \wedge \mathrm{critB} \wedge \neg\mathrm{critA}\}$$

cannot be true simultaneously.

4.3.3.2 Strong Correctness

It remains to be proven that all loops (except of course the outer ones) terminate, and hence the critical sections will be executed. We will prove termination for process A, with all arguments symmetrically applying to the proof of termination for process B.

Loop <A3>

1) Let $t=0$ iff prty=A and $t=1$ iff prty=B. Since prty is of type (A,B), $t\geq 0$ is invariant.

2) Since loop <A3> performs no actual computation, its steady-state weak parallel invariant is identical to its weak invariant.

$$\mathcal{J} = \neg\text{inA} \land \neg\text{critA} \land \text{critB} \Rightarrow \text{inB}$$

   Clearly $\mathcal{J} \land \text{prty=B} \Rightarrow t>0$.

3) The value of prty is affected only by statement <B7> in process B. Since prty←A corresponds to $t\leftarrow 0$, statement <B7> does not increase t.

4) If $\mathcal{J} \land \text{B}$ holds, then

$$\neg\text{inA} \land \neg\text{critA} \land \text{critB} \Rightarrow \text{inB} \land \text{prty=B}$$

   so $t=1$. Thus prty←A will decrease t to 0.

5) The only way that <B7> will not be executed is if loop <B1> does not terminate (assuming the critical section terminates). Since this loop terminates when inA is false, and since ¬inA is guaranteed by the steady-state invariant, non-termination of <B1> can only be a result of non-termination of an inner loop, namely <B3>. Since this loop depends on prty=A, and since prty=B is implied by $\mathcal{J} \land \text{B}$, <B3> must also terminate.

1-5 constitute the proof that loop <A3> (and hence loop <B3>) must always terminate.

Loop <A1>

1) Let $t=0$ iff ¬inB, and $t=1$ iff inB. Since inB is boolean, $t\geq 0$ is invariant.

2) The steady-state weak parallel invariant of loop <A1> is

$$\mathcal{J} = \text{inA} \land \neg\text{critA} \land \text{critB} \Rightarrow \text{inB} \land \text{prty=A}$$

   $\mathcal{J}$ is established after the first pass through the loop, and remains true thereafter. Clearly $\mathcal{J} \land \text{inB} \Rightarrow t>0$.

3) The value of t is potentially affected by <B0>, <B2>, <B4>, and <B6>. Statement <B4> cannot affect termination of loop <A1> because its precondition cannot hold at the same time as $\mathcal{J}$. Since inB is set to true by <B0>, that statement will not increase t unboundedly.

4) Statements <B2> and <B6> can potentially decrease t. However, because there exists a path from <B6> to a state in which inB is again true, we cannot depend on <B6> to cause termination. The existence of this path is reflected by the fact that all of the successive assertions between <B6> and <B0> can hold at the same time as $\mathcal{J} \wedge B$. Our only remaining hope is <B2>, and we see that there is no path from <B2> to a state in which inB is reset to true before <A1> terminates. This holds because

```
{$\mathcal{J} \wedge \neg inB \wedge \neg critB \wedge critA \Rightarrow inA$}
while prty=A do
      nothing
      od;
{false}
```

5) It remains to be shown that <B2> is guaranteed to be executed. Since the program consists of straight line code outside of loop <B1>, we are guaranteed to reach <B1> eventually. Since we can assume $\mathcal{J}$ holds, we are guaranteed to enter the loop, and since $\mathcal{J} \Rightarrow prty{=}A$, we must then execute <B2>.

1-5 constitute the proof that loop <A1> (and hence loop <B1>) must always terminate.

5. Verifying Restricted Parallel Programs

5.1. Introduction

In this chapter we take a decidedly different approach to parallel program correctness from those of Chapter 4. By restricting the syntax of parallel programs suitably, we can be much more precise about their semantics and more practical about automating a verification system.

5.2. Restrictions

To express the semantics of parallel programs precisely, we first need a working definition of what parallel program correctness means. The sequential case definition, termination in a particular state, will do only when we deal with terminating parallel programs. Consider parallel programs expressed in the syntax:

$$T\text{: } \underline{\text{cobegin}}\ E_1\ //\ E_2\ //\ \ldots\ //\ E_n\ \underline{\text{coend}}$$

and let us restrict the E_i to be single conditional critical regions [Brinch Hansen 73] of the form:

$$\underline{\text{with}}\ X_i\ \underline{\text{when}}\ B_i\ \underline{\text{do}}\ S_i\ \underline{\text{od}}$$

We can express the correctness of program T by requiring that it terminate (i.e. all processes must execute and terminate individually) in a state which satisfies some relation R.

Operating systems frequently contain non-terminating, cyclic processes, hence no final state exists. Consider programs of the form,

$$C\text{: } \underline{\text{cobegin}}\ \underline{\text{repeat}}\ E_1\ //\ \underline{\text{repeat}}\ E_2\ //\ \ldots\ //\ \underline{\text{repeat}}\ E_n\ \underline{\text{coend}}$$

where repeat <statement> causes infinite repetition of <statement>, and the E_i are as before. Then we can express the weak correctness of program C in terms of the achievement of a state satisfying some relation R_j, whenever process j executes S_j. We refer to this as weak correctness, since it may be that blocking develops (i.e. all processes become blocked at the same time, hence no further progress is made), or it may be that one or more processes deadlock (i.e. remain blocked forever with no possibility of continuing). Further, one or more processes may starve (i.e. remain blocked forever even though the possibility of continuing exists). Blocking, deadlock, and starvation are called strong correctness issues. No cyclic parallel program may be considered correct unless it is both weakly correct, so that we can describe its effect, and strongly correct (at least blocking-, deadlock-, and probably starvation-free), so that we can rely on the weak correctness effect being achieved.

We have recently learned of the work of van Lamsveerde and Sintzoff [Lamsveerde 76], which uses similar ideas in an effort to derive strongly correct programs. Our interest is in verification only.

5.3. Correctness of Terminating Parallel Programs

For program T, correctness amounts to the requirement that no matter in what order the processes begin to execute, all will execute (and terminate) eventually, and some relation R will hold when all are finished.

Let $\sigma(T)$ be the index set of processes in the cobegin-coend block T (i.e. $\sigma(T) = \{1, 2, \ldots, n\}$). Then

$$wp(T, R) = wp(\sigma(T), R)$$

where

$$wp(\phi, R) = R$$

$$wp(\Sigma, R) = (\exists j\in\Sigma)(B_j) \wedge (\forall j\in\Sigma)[B_j \Rightarrow wp(S_j, wp(\Sigma-\{j\}, R))]$$

This is justified as follows. We claim that $wp(\Sigma, R)$ has the interpretation, "No matter in what order the processes whose indices are contained in Σ execute, all will execute and terminate, with R established at the finish." We show this by induction on $|\Sigma|$.

For $|\Sigma|=0$, i.e. $\Sigma=\phi$, the interpretation clearly holds since $wp(\phi, R) = R$. Assume that the interpretation holds for $|\Sigma| = k$, $k\geq 0$. Now consider $\Sigma' = \Sigma \cup \pi$, where $\pi\notin\Sigma$. If $wp(\Sigma', R)$ is to have the proper interpretation, then it must guarantee that at least one of the processes listed in Σ' is runnable, else the program would make no further progress. Also, execution of any runnable process must terminate in a state which assures that the remaining processes will also execute and finally terminate with R. This is exactly what is expressed by $wp(\Sigma', R)$, since $|\Sigma'-\{j\}|=k$ means that $wp(\Sigma'-\{j\}, R)$ will cause the remaining processes all to execute and terminate with R by the induction hypothesis.

Finally, since $\sigma(T)$ contains all of the process indices of program T, $wp(\sigma(T), R)$ is the desired weakest pre-condition.

5.4. An Example

Consider the following program:

```
P:      cobegin
                with x when x=0 do x←2 od //
                with x when x=1 do x←0 od //
                with x when x=2 do x←1 od
        coend
```

Then $wp(P, x=a) = wp(\{1, 2, 3\}, x=a)$

$$\begin{aligned} = (x=0 \vee x=1 \vee x=2) & \\ & \wedge (x=0 \Rightarrow wp(x \leftarrow 2, wp(\{2, 3\}, x=a))) \\ & \wedge (x=1 \Rightarrow wp(x \leftarrow 0, wp(\{1, 3\}, x=a))) \\ & \wedge (x=2 \Rightarrow wp(x \leftarrow 1, wp(\{1, 2\}, x=a))) \end{aligned}$$

1) $wp(\{2, 3\}, x=a) =$

$$\begin{aligned} (x=1 \vee x=2) & \\ & \wedge (x=1 \Rightarrow wp(x \leftarrow 0, wp(\{3\}, x=a))) \\ & \wedge (x=2 \Rightarrow wp(x \leftarrow 1, wp(\{2\}, x=a))) \end{aligned}$$

$$\begin{aligned} = \quad (x=1 \vee x=2) & \\ & \wedge (x=1 \Rightarrow wp(x \leftarrow 0, x=2 \wedge a=1)) \\ & \wedge (x=2 \Rightarrow wp(x \leftarrow 1, x=1 \wedge a=0)) \end{aligned}$$

$$= \quad (x=1 \vee x=2) \wedge x \neq 1 \wedge (x=2 \Rightarrow a=0)$$

$$= \quad x=2 \wedge a=0$$

2) Similarly, $wp(\{1, 3\}, x=a) = (x=0 \wedge a=1)$

3) Similarly, $wp(\{1, 2\}, x=a) = (x=1 \wedge a=2)$

Hence wp({1, 2, 3}, x=a) =

$$\begin{aligned}&(x=0 \vee x=1 \vee x=2)\\ &\quad\wedge (x=0 \Rightarrow wp(x\leftarrow 2,\ x=2 \wedge a=0))\\ &\quad\wedge (x=1 \Rightarrow wp(x\leftarrow 0,\ x=0 \wedge a=1))\\ &\quad\wedge (x=2 \Rightarrow wp(x\leftarrow 1,\ x=1 \wedge a=2))\end{aligned}$$

$$= \quad (x=0 \vee x=1 \vee x=2) \wedge (x=0 \Rightarrow a=0) \wedge (x=1 \Rightarrow a=1) \wedge (x=2 \Rightarrow a=2)$$

$$= \quad x=a \wedge (a=0 \vee a=1 \vee a=2)$$

5.5. Correctness of Cyclic Processes

5.5.1 Weak Correctness

For programs of the form,

C: <u>cobegin</u> <u>repeat</u> E_1 // <u>repeat</u> E_2 // . . . // <u>repeat</u> E_n <u>coend</u>

as mentioned in section 5.2 we define weak correctness as the establishment of some *relation* R_j *whenever process j can execute.* Then

$$wlp(C, <R_1, R_2, \ldots, R_n>) = (\forall j \in 1..n)\ W_j(C, R_j)$$

is the weakest liberal pre-condition which guarantees this, where

$$W_j(C, R) = (\forall k \geq 0)\ W_j^k(C, R)$$

$$W_j^0(C, R) = B_j \Rightarrow R$$

$$W_j^{k+1}(C, R) = (\forall i \in 1..n)\ [B_i \Rightarrow wlp(S_i, W_j^k(C, R))]$$

W_j^k has the interpretation, "If there is some sequence of process executions of length k which results in B_j holding, then it must also result in R." This is clearly satisfied for k=0. Assume the interpretation of W_j^k holds for k=m, m≥0. Then, since W_j^{m+1} guarantees that any executable process will result in W_j^m, the interpretation must hold for all k≥0. The conjunction of W_j^k for all k≥0 then has the interpretation, "Any sequence of process executions which results in B_j holding will also result in R holding", thus whenever process j can run, R_j will be true.

5.5.2 Strong Correctness

5.5.2.1 Blocking

A set of processes is *blocked* if none of them can run, thus allowing no further progress. A cyclic parallel program is *blocking-free* if there is no execution sequence which leads to blocking. For program C, the weakest pre-condition which guarantees that C is blocking-free, denoted *wbp*(C), is given by

$$wbp(C) = (\forall k \geq 0)\ U_k(C)$$

where

$$U_0(C) = true$$

$$U_{k+1}(C) = (\exists j \in 1..n)(B_j) \wedge (\forall j \in 1..n)\ [B_j \Rightarrow wp(S_j, U_k(F))]$$

$U_k(C)$ has the interpretation, "At least k processes will execute." This is clearly satisfied for k=0. For $U_{k+1}(C)$, we must have that at least one process can execute, and that any process that does execute will result in $U_k(C)$, which is precisely what is expressed above. In the limit, $U_k(F)$ will guarantee an unbounded execution sequence, and hence that F is blocking-free.

5.5.2.2 Deadlock

If a cyclic parallel program reaches a state which forever excludes any possibility of a given process continuing, then that process is said to be *deadlocked*. A cyclic parallel program is *deadlock-free* if there is no execution sequence which leads to the deadlock of any process. The weakest pre-condition which guarantees program C to be deadlock-free, denoted *wdp*(C), is given by

$$wdp(C) = wbp(C) \wedge (\forall i \in 1..n)\ (\forall j \in 1..n)\ W_i(C, V(C, B_j)))$$

where W_i is as previously defined in section 5.5.1 and

$$V(C,R) = (\exists k \geq 0)\ V_k(R)$$

$$V_0(R) = R$$

$$V_{k+1}(R) = (\exists m \in 1..n)(B_m \wedge wp(S_m, V_k(R)))$$

$V(C, B_j)$ gives the weakest pre-condition that assures the existence of an execution sequence which establishes B_j. $W_i(C,V(C,B_j))$ thus gives the condition which guarantees the continued existence of such a sequence whenever process i is executable. The definition of wdp may be read, "All processes must leave invariant the condition which assures the possibility of any given process executing".

5.5.2.3 Starvation

A process is said to *starve* if it is prevented from running by virtue of the particular execution sequence taken by the parallel program, while in fact it would become runnable if some other execution sequence had been chosen. This is to be distinguished from deadlock, in which a state is reached from which no execution sequence can enable a given process. The weakest pre-condition guaranteeing non-starvation in a system of parallel processes, *wsp*(C), is given by

$$wsp(C) = wbp(C) \wedge (\forall i \in 1..n) \neg V(C, Z(\neg B_i))$$

where

$$Z(R) = (\forall k \geq 0)\ Z_k(R)$$

$$Z_0(R) = R$$

$$Z_{k+1}(R) = (\exists j \in 1..n)(B_j \wedge wp(S_j, Z_k(R)))$$

$Z(R)$ gives the weakest pre-condition that assures the existence of an unbounded execution sequence which maintains the truth of R. That is, $Z(\neg B_i)$ holds if and only if B_i is false and can be kept forever false by some execution sequence. $V(C, R)$ is as defined in the previous section, so $\neg V(C, Z(\neg B_i))$ guarantees that it is not possible to reach a state in which process i can be forever prevented from running.

5.5.3 Verification Methods

Having described the weakest pre-condition formalism for the class of programs with which we are concerned, we now discuss an approach to parallel program verification which is analogous to the "invariant relation" approach to sequential loop correctness (see section 4.3.2.2).

Theorem 5.1 (Non-blocking Invariant)

$$(\forall j\in 1..n)\ [\mathcal{I} \wedge B_j \Rightarrow wp(S_j, \mathcal{I})] \wedge [\mathcal{I} \Rightarrow (\exists j\in 1..n)B_j]$$

$$\vdash\ [\mathcal{I} \Rightarrow wbp(C)]$$

Proof

Rewriting the theorem, we must show

$$(\forall j\in 1..n)\ [\mathcal{I} \wedge B_j \Rightarrow wp(S_j, \mathcal{I})] \wedge [\mathcal{I} \Rightarrow (\exists j\in 1..n)B_j)] \wedge \mathcal{I} \Rightarrow (\forall k\geq 0)U_k$$

where

$$U_0 = \text{true}$$

$$U_{k+1} = (\exists j\in 1..n)(B_j) \wedge (\forall j\in 1..n)(B_j \Rightarrow wp(S_j, U_k))$$

We use induction on k.

1) For $k=0$, the theorem clearly holds.

2) Suppose $(\forall j\in 1..n)\ [\mathcal{I} \wedge B_j \Rightarrow wp(S_j, \mathcal{I})] \wedge [\mathcal{I} \Rightarrow (\exists j\in 1..n)B_j] \wedge \mathcal{I} \Rightarrow U_k$.

Then we will prove

$$(\forall j\in 1..n)\ [\mathcal{I} \wedge B_j \Rightarrow wp(S_j, \mathcal{I})] \wedge [\mathcal{I} \Rightarrow (\exists j\in 1..n)B_j] \wedge \mathcal{I} \Rightarrow U_{k+1}$$

That is,

$$(\forall j\in 1..n)\ [\mathcal{I} \wedge B_j \Rightarrow wp(S_j, \mathcal{I})] \wedge [\mathcal{I} \Rightarrow (\exists j\in 1..n)B_j] \wedge \mathcal{I} \wedge U_k \Rightarrow U_{k+1}$$

Substituting the definition of U_{k+1},

$$(\forall j\in 1..n)\ [\mathcal{I} \wedge B_j \Rightarrow wp(S_j, \mathcal{I})] \wedge [\mathcal{I} \Rightarrow (\exists j\in 1..n)B_j] \wedge \mathcal{I} \wedge U_k$$

$$\Rightarrow (\exists j\in 1..n)B_j \wedge (\forall j\in 1..n)\ [B_j \Rightarrow wp(S_j, U_k)]$$

Since we can derive $(\exists j\in 1..n)B_j$ from the hypotheses immediately, it remains to show

$$(\forall j\in 1..n)\ [\mathcal{I} \wedge B_j \Rightarrow wp(S_j, \mathcal{I})] \wedge [\mathcal{I} \Rightarrow (\exists j\in 1..n)B_j] \wedge \mathcal{I} \wedge U_k$$

$$\Rightarrow (\forall j\in 1..n)\ [B_j \Rightarrow wp(S_j, U_k)]$$

We can assume $\mathcal{I} \Rightarrow U_k$, since otherwise the implication clearly holds.

Then $(\forall j\in 1..n)\ [B_j \Rightarrow wp(S_j, \mathcal{I})] \Rightarrow (\forall j\in 1..n)\ [B_j \Rightarrow wp(S_j, U_k)]$

from the monotonicity of wp[8].

QED

Theorem 5.2 (Non-deadlock Invariant)

$$(\forall j\in 1..n)\ [\mathcal{I} \wedge B_j \Rightarrow wp(S_j,\mathcal{I})] \wedge [\mathcal{I} \Rightarrow (\exists j\in 1..n)B_j] \wedge (\forall j\in 1..n)\ [\mathcal{I} \Rightarrow V(C, B_j)]$$

$$\vdash\ [\mathcal{I} \Rightarrow wdp(C)]$$

[8] See Appendix A.

Theorem 5.3 (Non-Starvation Invariant)

$$(\forall j \in 1..n)\ [\mathcal{J} \wedge B_j \Rightarrow wp(S_j,\ \mathcal{J})] \wedge [\mathcal{J} \Rightarrow (\exists j \in 1..n) B_j] \wedge (\forall j \in 1..n)\ [\mathcal{J} \Rightarrow \neg Z(\neg B_j)]$$

$$\vdash\ [\mathcal{J} \Rightarrow wsp(C)]$$

The proofs of these theorems are sufficiently similar to that of the non-blocking theorem so as to make their inclusion unnecessary.

5.5.4 An Example

We shall prove that the following program is blocking-free using Theorem 5.1:

```
assert n>0 ∧ 0≤x≤n ∧ 0≤y≤n ∧ 0≤z≤n ∧ 0<x+y+z<3n;

     cobegin
          with (x,y) when x>0 ∧ y<n do x←x-1; y←y+1 od //
          with (y,z) when y>0 ∧ z<n do y←y-1; z←z+1 od //
          with (z,x) when z>0 ∧ x<n do z←z-1; x←x+1 od
     coend
```

This program models the action of three processes moving data among three buffers, e.g.

```
     cobegin
          q←remove(b1); insert(f(q),b2) //
          r←remove(b2); insert(g(r),b3) //
          s←remove(b3); insert(h(s),b1)
     coend
```

Let us take the input assertion to be our invariant $\mathcal{J}$. Then we must prove,

1) $(\forall j \in 1..n)\ [\mathcal{J} \wedge B_j \Rightarrow wp(S_j, \mathcal{J})]$

2) $\mathcal{J} \Rightarrow (\exists j \in 1..n)\ B_j$

Proof

1.1) $\mathcal{J} \wedge B_1 \Rightarrow wp(S_1, \mathcal{J})$

$$n>0 \wedge 0 \leq x \leq n \wedge 0 \leq y \leq n \wedge 0 \leq z \leq n \wedge 0<x+y+z<3n \wedge x>0 \wedge y<n$$
$$\Rightarrow wp(x \leftarrow x-1;\ y \leftarrow y+1,\ \mathcal{J})$$

$$n>0 \wedge 0<x \leq n \wedge 0 \leq y<n \wedge 0 \leq z \leq n \wedge 0<x+y+z<3n$$
$$\Rightarrow n>0 \wedge 1 \leq x \leq n+1 \wedge -1 \leq y \leq n-1 \wedge 0 \leq z \leq n \wedge 0<x+y+z<3n$$

1.2) $\mathcal{J} \wedge B_2 \Rightarrow wp(S_2, \mathcal{J})$ by symmetry

1.3) $\mathcal{J} \wedge B_3 \Rightarrow wp(S_3, \mathcal{J})$ by symmetry

2) $\mathcal{J} \Rightarrow (\exists j \in 1..n)\ B_j$

$$n>0 \wedge 0 \leq x \leq n \wedge 0 \leq y \leq n \wedge 0 \leq z \leq n \wedge 0<x+y+z<3n$$
$$\Rightarrow (x>0 \wedge y<n) \vee (y>0 \wedge z<n) \vee (z>0 \wedge x<n)$$

$$(x \leq 0 \vee y \geq n) \wedge (y \leq 0 \vee z \geq n) \wedge (z \leq 0 \vee x \geq n)$$
$$\Rightarrow n \leq 0 \vee x<0 \vee x>n \vee y<0 \vee y>n \vee z<0 \vee z>n \vee x+y+z \leq 0 \vee x+y+z \geq 3n$$

$$(x \leq 0 \wedge y \leq 0 \wedge z \leq 0) \vee (x \leq 0 \wedge y \leq 0 \wedge x \geq n) \vee (x \leq 0 \wedge z \geq n \wedge z \leq 0)$$
$$\vee (x \leq 0 \wedge z \geq n \wedge x \geq n) \vee (y \geq n \wedge y \leq 0 \wedge z \leq 0) \vee (y \geq n \wedge y \leq 0 \wedge x \geq n)$$
$$\vee (y \geq n \wedge z \geq n \wedge z \leq 0) \vee (y \geq n \wedge z \geq n \wedge x \geq n)$$
$$\Rightarrow n \leq 0 \vee x<0 \vee x>n \vee y<0 \vee y>n \vee z<0 \vee z>n \vee x+y+z \leq 0 \vee x+y+z \geq 3n$$

which can be seen to be true by inspection.

6. Conclusions

In this last chapter we shall attempt to draw the thesis together in a coherent fashion. Section 6.1 contains a summary of the material presented in each chapter. Section 6.2 describes what we think are the contributions made by the thesis. In section 6.3 we present our evaluation of the feasibility of operating system verification. Section 6.4 describes what we regard as important areas for future research.

6.1. Summary

In Chapter 2 we presented a methodology for the design, specification, implementation, and verification of highly modular programs. The methodology relies upon the concept of an abstract data type to provide well-defined module boundaries which, by virtue of their encapsulation of low-level data structure access, also conveniently encapsulate the verification of data structure consistency. Based upon the comparison of several specification techniques, one was chosen which we think provides the most intuitive means of transforming ideas into mathematical form during the process of formal specification. The example of a queue was used in each case to provide a basis for comparison. Once a decision was made as to the nature of formal specifications, we described a method of verifying their consistency by proving the invariance of certain properties of the abstract state.

Chapter 3 expanded upon the methodology of Chapter 2 in order to investigate its applicability to the design and verification of operating systems. The relationship between the structure of a system and the language in which it is implemented was explored. If the system is hierarchically structured, then the implementation language used to construct one level is a combination of that used to implement the level below and the facilities changed by the level below.

Chapter 3 also contains a large example - the design, specification, implementation, and verification of the process dispatcher of a hypothetical system. Particular emphasis was placed on verifying the specifications in addition to the implementation. Using the proof technique outlined in Chapter 2, properties such as "the current process has highest priority" and "dispatching is done in Round Robin fashion" were shown to be true of any implementation which correctly models the specifications.

Because operating systems employ much concurrency in their implementation, we devoted the next two chapters to various approaches to verifying the total correctness of parallel programs. In Chapter 4 we explored two of these, one using Dijkstra's weakest pre-condition semantics [Dijkstra 76], and the other using the axiomatic weak correctness approach of Owicki [Owicki 75]. Both were applied to a general class of parallel programs.

The weakest pre-condition approach to verifying general parallel programs led to the requirement for much more rigorously specified semantics of sequential control constructs, because these are usually made up of several primitive machine instructions, after any one of which pre-emption may occur. The result was a complete but computationally difficult proof technique for the parallel control construct.

Owicki's axiomatic approach is less algorithmic, because it requires the insightful addition of auxiliary variables. However, it is more feasible computationally. Also in Chapter 4 then, we presented our own interpretation of her methodology, including the principle of non-interference, and discussed an approach to verifying termination which generalized Dijkstra's approach for the sequential case [Dijkstra 76].

In Chapter 5, we returned to the weakest pre-condition approach, but applied it to a restricted class of parallel programs. The restrictions allowed us to formally define the weakest pre-conditions which guarantee both weak correctness and the absence of blocking, deadlock, and starvation. Although these definitions are complex, we were able to formulate theorems which make use of the invariant relation concept in order to guarantee correctness. Several examples were presented.

6.2. Contributions of the Thesis

Following are what we consider to be the contributions made by this research. The order in which they are presented is not intended to convey any idea of relative importance, but is based more or less sequentially upon the material presented.

1) The idea of developing an implementation language hierarchically, based upon the structure of the system it is being used to implement.

2) The idea of casting the correctness of formal specifications in terms of invariants which relate the data types involved, and the method of verifying these invariants. This includes the notion that such abstract ideas as "fairness" can be verified of the specifications if they can be expressed in terms of these invariants.

3) The fact that the design, implementation, and verification of a small but fairly realistic part of an operating system, the process dispatcher, was shown to be relatively straightforward using the proposed methodology.

4) The predicate transformational approach to parallel program correctness, most importantly the characterization of weakest pre-conditions for weak

correctness, absence of blocking, deadlock, and starvation, and the formulation of *invariant relation* theorems for those concepts.

5) The extension of the weak correctness methodology of Owicki [Owicki 75] to include the proof of arbitrary parallel program loop termination. This includes the generalization of Dijkstra's methodology for sequential loop termination [Dijkstra 76] and the concept of a "steady-state" loop invariant.

6.3. On the Feasibility of Operating Systems Verification

After expending a good deal of effort on the research described in the thesis, and in light of our resulting experience, it would seem appropriate that we comment upon the possibility of carrying out the verification of an entire operating system.

The primary problem we encountered during the process of carrying out program proofs was the inappropriateness of pencil and paper to the task. Much of the work involved was tedious, requiring continuous re-copying of non-trivial assertions in order to effect the step-by-step transformations dictated by weakest pre-condition computation. This particular process, usually called verification condition (VC) generation, is undoubtedly best handled automatically by a computer program, and indeed numerous implementations of such programs exist, for example [Suzuki 75, Good 75]. We made a conscious decision neither to implement nor to use an existing VC generator, principally because we desired the freedom of not fixing a syntax for our implementation language. Since for any real attempt at verification of an entire operating system we could easily write a VC generator, much of this tedium should disappear in practice.

There is no question that a powerful automatic theorem proving system is also needed. Most of the quantifier-free verification conditions could be proved rather easily by a relatively simple deductive system, but owing to the nature of the abstract data type mechanism, assertions which are quantified over types are very common. For example, an invariant which describes doubly-linked circular lists of objects of type T is

$$(\forall x \in T)\ (x.succ.pred = x \ \wedge \ x.pred.succ = x)$$

The proof of the procedure appendR of type Multiq in Chapter 3 relies very heavily on this invariant, and no simple-minded theorem prover can handle it [9]. In our opinion however, a

[9] We have in fact tried out the proof of a PASCAL version of the procedure on both Suzuki's verifier [Suzuki 75] and the ISI verifier [Good 75] without success. This is not to say that some other equivalent definition of correctness couldn't be used productively in this case, but we wish to stick to our comments about specification techniques in Chapter 2.

highly interactive theorem prover could substitute human intelligence for a sophisticated automatic deduction system with great success, since after all very few candidate theorems derived from computer programs require continuous creativity to be proven.

In addition to the VC generator and interactive theorem prover, a VC generator for specifications, as discussed in section 2.3, should be part of any system used to verify large-scale programs. Furthermore, the entire verification system must be part of a total system development environment in which there exist representations of inter-module dependencies. This bookkeeping system must have the knowledge to invalidate and regenerate the proofs of modules which are potentially affected by changes to other modules. Because of the sheer size of this sort of information in a real system, there is no hope of ever remembering such dependencies without machine assistance. An attempt along these lines, although for the moment primarily concerned with documentation and inter-module version consistency, is described in [Habermann 77]. Work is also in progress to formalize both inter- and intra-module dependencies [Cooprider 77], and this should serve as a basis for the exploration of a development and maintenance environment for correct programmed systems.

It is not clear that the method of describing data structures by type-quantified invariants as we have used them is sufficient. A major problem is the use of recursive functions in these assertions. For example, a binary tree may be described by the invariant

$$(\forall x \in node)\ btree(x)$$

where $btree(x) = \text{if } x = null \text{ then } true \text{ else } btree(x.left) \wedge btree(x.right)$.

The function btree is only partial if there is a path from any node to itself. Dealing with such assertions is quite complicated, and it remains to be seen if the data structures used by an operating system can all be simple enough to avoid these complications.

Finally, our parallel program correctness work is only a basis for future work. Since no real operating system will use conditional critical regions because of their inherent implementation inefficiency, more work must be done on using the critical region formal basis to deal with more practical synchronization primitives. Some of this has lately been done, and is described in [Flon 77]. An introduction to various synchronization primitives and their verification may be found in [Andler 77].

The conclusion we draw from this evaluation of the feasibility of operating system verification is that it hinges upon the development of a powerful programming and verification environment and the further investigation of parallel program correctness as described above. There is every reason to believe that five years to a toy environment and system, and ten to a practical environment and system are fair estimates.

6.4. Areas of Importance for Future Research

As mentioned in the preceding section, there remain several important and interesting problems to be solved before we can hope to achieve a verified operating system. We list what we consider to be the three most important.

1) Can the process of verifying the correctness of procedures which manage complex data structures be made simpler by judicious choice of the consistency invariant, or is some other method of specification needed?

2) What should be the attributes of an environment for the development and maintenance of correct programs? What should the user interface to an interactive program verification system be? Can program verification be done incrementally? That is, can a modified program be re-verified only insofar as is absolutely necessary?

3) How can we apply the parallel program correctness ideas of Chapters 4 and 5 to practical synchronization mechanisms? There has been work done on proving the correctness of concurrently used abstract data types [Howard 76, Flon 76, Owicki 77] but very little has been said about the correctness of processes which use those data types. What effect do monitors and path expressions have on the provability of absence of deadlock? (Campbell has done an investigation of this question for highly simplified path expressions [Campbell 76].)

The area of large-scale program construction, maintenance, and verification should provide a rich source of interesting research questions over the years to come.

Appendix A: Weakest Pre-Condition Semantics

Definition

The *weakest pre-condition* of a statement S and predicate R, wp(S, R) is the necessary and sufficient condition which guarantees that S will terminate leaving R true.

1) $wp(\text{<empty>}, R) = R$

2) $wp(x \leftarrow e, R) = R|_{e}^{x}$

All free occurrences of x in R are replaced by e.

3) $wp(\underline{\text{if}}\ B\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}, R) =$

$$[B \Rightarrow wp(S_1, R)] \wedge [\neg B \Rightarrow wp(S_2, R)]$$

4) $wp(S_1;S_2, R) = wp(S_1, wp(S_2, R))$

5) $wp(\underline{\text{while}}\ B\ \underline{\text{do}}\ S\ \underline{\text{od}}, R) = (\exists k)\ G_k$

where $G_0 = \neg B \wedge R$

$G_{k+1} = B \wedge wp(S, G_k)$

Definition

The *weakest liberal pre-condition* of a statement S and predicate R, wlp(S, R) is the necessary and sufficient condition which guarantees that S will either terminate leaving R true or not terminate at all.

$$wp(S, R) = wlp(S, R) \wedge wp(S, true)$$

An important property of the weakest pre-condition is *monotonicity*, which is expressed by the axiom

$$P \Rightarrow Q \vdash wp(S, P) \Rightarrow wp(S, Q)$$

Since wp(S, P) characterizes the set of states which guarantee the truth of P after S is executed, any one of those states must also guarantee the truth of Q after S as long as $P \Rightarrow Q$.

Weakest pre-condition has a dual, called *strongest post-condition*.

Definition

The *strongest post-condition* of a statement S and predicate R, sp(S, R), is the strongest possible characterization of the set of states which can hold after S terminates, given that it is started with R holding.

1) $sp(\text{<empty>}, R) = R$

2) $sp(x \leftarrow e, R|^{x}_{e}) = R$

3) $sp(\underline{\text{if}}\ B\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}, R) =$

$$[R \wedge B \Rightarrow sp(S_1, R)] \wedge [R \wedge \neg B \Rightarrow sp(S_2, R)]$$

4) $sp(S_1; S_2, R) = sp(S_2, sp(S_1, R))$

5) $sp(\underline{\text{while}}\ B\ \underline{\text{do}}\ S\ \underline{\text{od}}, R) = \neg B \wedge (\exists k)\ F_k$

where $F_0 = R$

$F_{k+1} = sp(S, F_k \wedge B)$

Appendix B: Primitive Abstract Data Types

The definitions we present are in the style of [Hoare 72a].

Notation

X:t means X has type t.

Enumeration Types

1) The type $(d_1, d_2, \ldots d_n)$ where $n \geq 1$ and the d_j are distinct identifiers is called an *enumeration type*.

2) If $x:(d_1, d_2, \ldots d_n)$ then $(\exists j \in 1..n)\ x = d_j$.

3) $(\forall j \in 1..n)\ d_{j+1} > d_j$

Range Types

A *range type* is an enumeration type, denoted a..b, whose elements are a contiguous subset (from a to b) of the elements of another enumeration type, e.g. 1..n is a range type of the integers.

Records

1) If $a_1{:}t_1, a_2{:}t_2, \ldots a_n{:}t_n$ for any types t_j, then

$$(a_1, a_2, \ldots a_n) : (f_1{:}t_1, f_2{:}t_2, \ldots f_n{:}t_n)$$

for any identifiers f_j.

2) $x:(f_1{:}t_1, f_2{:}t_2, \ldots f_n{:}t_n)$ and $x=(a_1, a_2, \ldots a_n)$ iff $x.f_1=a_1, x.f_2=a_2, \ldots x.f_n=a_n$.

3) if $x:(f_1{:}t_1, f_2{:}t_2, \ldots f_n{:}t_n)$ and $x=(a_1, a_2, \ldots a_n)$ then

$$\langle x, f_j, k\rangle.f_m = \text{if } j=m \text{ then } k \text{ else } x.f_m$$

Vectors

The rules for vectors are taken from [Luckham 76].

1) If D is an enumeration type and R is any type, then vector D of R is a *vector type.*

Let X:vector D of R.

2) $(\forall d \in D)\ (X[d] \in R)$

3) $<X, d, r>[d'] = \text{if } d'=d \text{ then } r \text{ else } X[d']$

4) $j = (>k)P(X[k]) \equiv P(X[j]) \wedge (\forall k>j)\neg P(X[k])$

$j= (<k)P(X[k]) \equiv P(X[j]) \wedge (\forall k<j)\neg P(X[k])$

Sequences

1) If R is any type, then sequence of R is a *sequence type.*

Let X:sequence of R.

2) If $r \in R$ then either

   2.1) $X=r$
   2.2) $X=\lambda$ (a distinguished symbol)
   2.3) $X=r \sim Y$ where Y:sequence of R

3) first(x): sequence of R → R

   3.1) if $X=r$ then r
   3.2) if $X=\lambda$ then undefined
   3.3) if $X=r \sim Y$ then r

4) has(X,k): sequence of R x R → boolean

   4.1) if $X=r$ then $(r=k)$
   4.2) if $X=\lambda$ then false
   4.3) if $X=r \sim Y$ then $(r=k) \vee \text{has}(Y,k)$

5) length(X): sequence of R → integer

```
    5.1) if X=r then 1
    5.2) if $X=\lambda$ then 0
    5.3) if X=r~Y then 1+length(Y)
```

6) equal(X,Z): sequence of R x sequence of R → boolean

```
    if $X=\lambda \wedge Z=\lambda$ then true
    else if $X=r \wedge Z=s$ then (r=s)
    else if $\text{length}(X)\neq\text{length}(Z)$ then false
    else $(X=r\sim X_1) \wedge (Z=s\sim Z_1) \wedge (r=s) \wedge \text{equal}(X_1,Z_1)$
```

References

[Ambler 77] Ambler, A. L. et al, GYPSY: A Language for Specification and Implementation of Verifiable Programs. Proceedings of an ACM Conference on Language Design for Reliable Software, *SIGPLAN Notices 12,3* (March 1977), 1-10.

[Andler 77] Andler, S., Synchronization Primitives and the Verification of Concurrent Programs. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (May 1977).

[Bell 71] Bell, C. G. and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, 1971.

[Brinch Hansen 70] Brinch Hansen, P., The Nucleus of a Multiprogramming System. *Comm. ACM 13,4* (April 1970), 238-241.

[Brinch Hansen 73] Brinch Hansen, P., Operating Systems Principles, Prentice Hall, 1973.

[Campbell 76] Campbell, R. H., Path Expressions: A Technique for Specifying Process Synchronization. Ph. D. Thesis, University of Newcastle Upon Tyne (1976).

[Cooprider 77] Cooprider, L., Ph. D. Thesis (in progress). Carnegie-Mellon University (1977).

[Dahl 68] Dahl, O. J. et al, Simula 67 Common Base Language. Norwegian Computing Center, Oslo (May 1968).

[Dijkstra 65] Dijkstra, E. W., Solution of a Problem in Concurrent Programming Control. *Comm. ACM 8,9* (Sept. 1965), 569.

[Dijkstra 68] Dijkstra, E. W., The Structure of the T.H.E. Multiprogramming System. *Comm. ACM 11,5* (May 1968), 341-346.

[Dijkstra 76] Dijkstra, E. W., A Discipline of Programming, Prentice Hall, 1976.

[Flon 74] Flon, L., A Survey of Some Issues Concerning Abstract Data Types. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (Sept. 1974).

[Flon 75] Flon, L., Program Design With Abstract Data Types. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (June 1975).

[Flon 76] Flon, L. and Habermann, A. N., Towards the Construction of Verifiable Software Systems. Proceedings of the ACM Conference on Data: Abstraction, Definition and Structure, *SIGPLAN Notices 8,2* (March 1976), 141-148.

[Flon 77] Flon, L. and Suzuki, N., Nondeterminism and the Correctness of Parallel Programs. Proc. IFIP Working Conference on the Formal Description of Programming Concepts, New Brunswick, Canada (to appear Aug. 1977).

[Floyd 67] Floyd, R. W., Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Vol. 19*, J. T. Schwartz (ed.), American Mathematical Society, Providence, R.I., 1967, 19-32.

[Good 75] Good, D. I. et al, An Interactive Program Verification System. Proceedings of the International Conference on Reliable Software, *SIGPLAN Notices* (June 1975), 482-492.

[Geschke 77] Geschke, C. M., et al, Early Experience With Mesa. Proceedings of an ACM Conference on Language Design for Reliable Software (abstract only, to appear Comm. ACM), *SIGPLAN Notices 12,3* (March 1977), 138.

[Griffiths 74] Griffiths, P., SYNVER: A System for the Automatic Synthesis and Verification of Synchronization Processes. Harvard University (1974).

[Guttag 75] Guttag, J. V., The Specification and Application to Programming of Abstract Data Types. Ph.D. Thesis, University of Toronto (Sept. 1975).

[Guttag 77] Guttag, J. V. et al, Some Extensions to Algebraic Specifications. Proceedings of an ACM Conference on Language Design for Reliable Software, *SIGPLAN Notices 12,3* (March 1977), 63-67.

[Habermann 72] Habermann, A. N., Synchronization of Communicating Processes. *Comm. ACM 15,3* (March 1972), 171-176.

[Habermann 76] Habermann, A. N., Flon, L., and Cooprider, L., Modularization and Hierarchy in a Family of Operating Systems. *Comm. ACM 19,5* (May 1976), 266-272.

[Habermann 77] Habermann, A. N., On System Development Control. to appear (1977).

[Hoare 69] Hoare, C. A. R., An Axiomatic Basis for Computer Programming. *Comm. ACM 12,10* (Oct. 1969), 576-580.

[Hoare 71a] Hoare, C. A. R., Towards a Theory of Parallel Programming. In *Operating Systems Techniques*, Hoare and Perrot (eds.), Academic Press, London, 1971.

[Hoare 71b] Hoare, C. A. R., Procedures and Parameters: An Axiomatic Approach. In *Symposium on Semantics of Algorithmic Languages*, E. Engeler (ed.), Springer Verlag, 1971, 102-116.

[Hoare 72a] Hoare, C. A. R., Notes on Data Structuring. In *Structured Programming*, Dahl, Dijkstra and Hoare, Academic Press, London, 1972.

[Hoare 72b] Hoare, C. A. R., Proof of Correctness of Data Representations. *Acta Informatica 1* (1972).

[Hoare 74] Hoare, C. A. R., Monitors: An Operating System Structuring Concept. *Comm. ACM 17,10* (Oct. 1974), 549-557.

[Howard 76] Howard, J. H., Proving Monitors. *Comm. ACM 19,5* (May 1976), 273-279.

[Johnson 76] Johnson, R. T. and Morris, J. B., Abstract Data Types in the Model Programming Language. Proceedings of the ACM Conference on Data: Abstraction, Definition and Structure, *SIGPLAN Notices 8,2* (March 1976), 36-46.

[Karp 76] Karp, R. A. and Luckham, D. C., Verification of Fairness in an Implementation of Monitors. Proc. 2nd International Conference on Software Engineering (Oct. 1976), 40-46.

[Lamsveerde 76] van Lamsveerde, A. and Sintzoff, M., Formal Derivation of Strongly Correct Parallel Programs. MBLE Research Report, Brussels, Belgium (1976).

[Liskov 72] Liskov, B., The Design of the VENUS Operating System. *Comm. ACM 15,3* (March 1972), 144-149.

[Liskov 74] Liskov, B. and Zilles, S., Programming With Abstract Data Types. *SIGPLAN Notices* (April 1974), 50-59.

[Liskov 76] Liskov, B. and Zilles, S., Specification Techniques for Data Abstractions. Proceedings of the International Conference on Reliable Software, *SIGPLAN Notices* (June 1976), 72-87.

[Luckham 76] Luckham, D. and Suzuki, N., Automatic Program Verification V: Verification-Oriented Proof Rules for Arrays, Records and Pointers. Stanford University memo AIM-278 (March 1976).

[Neumann 74] Neumann, P. G. et al, On the Design of a Provably Secure Operating System. Proc. IRIA Workshop on Protection in Operating Systems, Paris (Aug. 1974), 161-175.

[Organick 72] Organick, E. I., The Multics System: An Examination of its Structure, MIT Press, 1972.

[Owicki 75] Owicki, S., Axiomatic Proof Techniques for Parallel Programs. Ph.D. Thesis, Department of Computer Science, Cornell University (July 1975).

[Owicki 76] Owicki, S. and Gries, D., Verifying Properties of Parallel Programs: An Axiomatic Approach. *Comm. ACM 19,5* (May 1976), 279-285.

[Owicki 77] Owicki, S., Verifying Concurrent Programs with Shared Data Classes. Proc. IFIP Working Conference on the Formal Description of Programming Concepts, New Brunswick, Canada (to appear Aug. 1977).

[Parnas 72a] Parnas, D. L., A Technique for Software Module Specification With Examples. *Comm. ACM 15,5* (May 1972), 330-336.

[Parnas 72b] Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACM 15,12* (Dec. 1972), 1053-1058.

[Parnas 74] Parnas, D. L., On a 'Buzzword': Hierarchical Structure. Proceedings of the IFIPS Congress 74, Stockholm, Sweden (1974).

[Popek 77] Popek, G. J. et al, Notes on the Design of Euclid. Proceedings of an ACM Conference on Language Design for Reliable Software, *SIGPLAN Notices 12,3* (March 1977), 11-18.

[Prenner 72] Prenner, C. J., Multi-Path Control Structures for Programming Languages. Ph.D. Thesis, Harvard University (May 1972).

[Saxena 76] Saxena, A. R., A Verified Specification of a Hierarchical Operating System. Ph.D. Thesis, Stanford University (Jan. 1976).

[Schaffert 75] Schaffert, C., Snyder, A., and Atkinson, R., The CLU Reference Manual. MAC-TR, MIT (June 1975).

[Schroeder 75] Schroeder, M., Engineering a Security Kernel for Multics. Proc. Fifth SIGOPS Symposium on Operating Systems Principles, *Operating Systems Review 9,5* (Nov. 1975), 25-32.

[Suzuki 75] Suzuki, N., Verifying Programs by Algebraic and Logical Reduction. Proceedings of the International Conference on Reliable Software, *SIGPLAN Notices* (June 1975), 473-481.

[Wegbreit 76] Wegbreit, B. and Spitzen, J. M., Proving Properties of Complex Data Structures. *Journal ACM 23,2* (April 1976), 389-396.

[Wulf 74] Wulf, W. A. et al, HYDRA: The Kernel of a Multiprocessor Operating System. *Comm. ACM 17,6* (June 1974), 337-345.

[Wulf 75] Wulf, W. A. et al, The Design of an Optimizing Compiler, American Elsevier, 1975.

[Wulf 76] Wulf, W. A., London, R., and Shaw, M., Abstraction and Verification in ALPHARD. In *New Directions in Algorithmic Languages*, S. A. Schuman (ed.), IRIA, France, 1976, 217-295.